antara-gaming-sdk

Release 1.2.0

Roman Sztergbaum Tolg Ay

Apr 02, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

Welcome to the documentation of the antara-gaming-sdk. `antara-gaming-sdk` is an SDK programmed in **C++ 17** that aims to be very fast in runtime by using the maximum of features at the compilation while remaining extensible to the runtime. It is based on an architecture of *modules* that can be used separately or together.

Below a list of topics covered by the documentation:

1.1 Antara Gaming API documentation

1.1.1 antara::gaming::animation2d

1.1.2 antara::gaming::config

template<typename **TConfig**>
TConfig `antara::gaming::config::`**`load_configuration`**(boost::filesystem::path &&*config_path*, std::string *filename*)

This function allows us to load a configuration through a `path` and `filename`. There are three different behaviors in this function:

- if the parameter path does not exist the function will attempt to create the directories of the given `path`.
- if the configuration does not exist a default one will be **created**.
- if the `path` and the `name` of the file exists, the contents of the configuration will be **loaded**.

Example:

```
auto cfg = config::load_configuration<my_game::config>(std::filesystem::current_
↪path() / "assets/config", "my_game.config.json");
```

Return a loaded/created configuration.

Template Parameters

- `TConfig`: the type of template you want to load

Parameters

- `config_path`: the path to the configuration you want to load
- `filename`: the name of the configuration you want to load.

1.1.3 antara::gaming::core

constexpr const char *antara::gaming::**version**()
Function that allows us to find the current version of the SDK.

Example:

```
#include <iostream>
#include <antara/gaming/core/version.hpp>

void print_version() {
    std::cout << antara::gaming::version() << std::endl;
}
```

Return the current version of the SDK as a `const char *`

Note: The result of this function can be deduced at compile-time.

1.1.4 antara::gaming::ecs

antara::gaming::ecs::system_manager

class system_manager
This class allows the manipulation of systems, the addition, deletion, update of systems, deactivation of a system, etc.

Public Functions

system_manager(entt::registry &*registry*, bool *subscribe_to_internal_events* = true)
Constructor.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager mgr{entity_registry};
}
```

Parameters

- `registry`: The entity_registry is provided to the system when it is created.
- `subscribe_to_internal_events`: Choose whether to subscribe to default *system_manager* events

Note: Principal Constructor.

~system_manager()
Destructor.

void **receive_add_base_system**(**const** ecs::event::add_base_system &*evt*)
Public member functions.

Parameters

- `evt`: The event that contains the system to add

void **start**()
This function tells the system manager that you start your game.

Note: This function, which indicates the game is spinning, allows actions to be done at each end of the frame like delete systems or add them while

we are going to iterate on

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    system_manager.start();
    return 0;
}
```

std::size_t **update**()

Note: This is the function that update your systems. Based on the logic of the different kinds of antara systems, this function takes care of updating your systems in the right order.

Warning: If you have not loaded any system into the system_manager the function returns 0. If you decide to mark a system, it's automatically deleted at the end of the current loop tick through this function. If you decide to add a system through an *ecs::event::add_base_system event*, it's automatically added at the end of the current loop tick

through this function.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    system_manager.start();
    // ... added 5 differents systems here
    std::size_t nb_systems_updated = system_manager.update();
    if (nb_systems_updated != 5) {
       // Oh no, i expected 5 systems to be executed in this game loop tick
    }
    return 0;
}
```

Return number of systems which are successfully updated

std::size_t **update_systems**(*system_type system_type_to_update*)

Note: This function is called multiple times by update(). It is useful if you want to program your own update function without going through the one provided by us.

Return number of systems which are successfully updated

See *update*

Parameters

- `system_type_to_update`: kind of systems to update (pre_update, logic_update, post_update)

template<typename **TSystem**>
const *TSystem* &**get_system**() **const**

This function allows you to get a system through a template parameter.

Return A reference to the system obtained.

Template Parameters

- `TSystem`: represents the system to get.

template<typename **TSystem**>
TSystem &**get_system**()

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

```

(continues on next page)

(continued from previous page)

```
int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    system_manager.start();
    // ... added 2 differents systems here (render_system, and a log_system)
    auto& render_system = system_manager.get_system<game::render_system>();

    const auto& log_system = system_manager.get_system<game::log_system>();
    return 0;
}
```

template<typename ...**TSystems**>
std::tuple<std::add_lvalue_reference_t<*TSystems*>...> **get_systems**()
This function allow you to get multiple system through multiple templates parameters.

Note: This function recursively calls the get_system function Based on the logic of the different kinds of antara systems, this function takes care of updating your systems in the right order.

Return Tuple of systems obtained.

See *get_system*

Template Parameters

- TSystems: represents a list of systems to get

template<typename ...**TSystems**>
std::tuple<std::add_lvalue_reference_t<std::add_const_t<*TSystems*>>...> **get_systems**() **const**
const version overload of get_systems

Note: This function is marked as nodiscard.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    // ... added differents systems here
    // Called from a const context
    auto &&[system_foo, system_bar] = system_manager.get_systems<system_foo,␣
↪system_bar>();

    // Called from a non const context
    auto&&[system_foo_nc, system_bar_nc] = system_manager.get_systems<system_
↪foo, system_bar>();
```

(continues on next page)

(continued from previous page)

```
    // Get it as a tuple
    auto tuple_systems = system_manager.get_systems<system_foo, system_bar>();
    return 0;
}
```

See *get_systems*

template<typename **TSystem**>
bool **has_system**() **const**

This function allow you to verify if a system is already registered in the *system_manager*.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.has_system<my_game::render_system>();
    if (!result) {
        // Oh no, i don't have a rendering system.
    }
    return 0;
}
```

Template Parameters

- `TSystem`: Represents the system that needs to be verified

Note: This function is marked as nodiscard.

Return true if the system has been loaded, false otherwise

template<typename ...**TSystems**>
bool **has_systems**() **const**

This function allow you to verify if a list of systems is already registered in the *system_manager*.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
```

(continues on next page)

(continued from previous page)

```
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.has_systems<my_game::render_system, my_
↪game::input_systems>();
    if (!result) {
        // Oh no, atleast one of the systems is not present
    }
    return 0;
}
```

See *has_system*

Note: This function is marked as nodiscard. This function recursively calls the has_system function.

Template Parameters

- `TSystems`: represents a list of system that needs to be verified

Return true if the list of systems has been loaded, false otherwise

template<typename **`TSystem`**>
bool **`mark_system`**`()`

This function marks a system that will be destroyed at the next tick of the game loop.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.mark_system<my_game::render>();
    if (!result) {
        // Oh no the system has not been marked.
        // Did you mark a system that is not present in the system_manager ?
    }
    return 0;
}
```

Template Parameters

- `TSystem`: Represents the system that needs to be marked

Return true if the system has been marked, false otherwise

template<typename ...**`TSystems`**>
bool **`mark_systems`**`()`

This function marks a list of systems, marked systems will be destroyed at the next tick of the game loop.

Note: This function is marked as nodiscard. This function recursively calls the mark_system function.

Template Parameters

- `TSystems`: Represents a list of systems that needs to be marked

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.mark_systems<my_game::render, my_game::input>
↪();
    if (!result) {
        // Oh no, atleast one of the system has not been marked.
        // Did you mark a system that is not present in the system_manager ?
    }
    return 0;
}
```

Return true if the list of systems has been marked, false otherwise

See *mark_system*

template<typename **TSystem**>
bool **enable_system**()

This function enable a system.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.enable_system<my_game::input>();
    if (!result) {
        // Oh no, this system cannot be enabled.
        // Did you enable a system that is not present in the system_manager ?
    }
    return 0;
}
```

Template Parameters

- `TSystem`: Represents the system that needs to be enabled.

Return true if the system has been enabled, false otherwise

template<typename ...**TSystems**>
bool **enable_systems**()
This function enable a list of systems.

Note: This function recursively calls the enable_system function

---

Template Parameters

- `TSystems`: Represents a list of systems that needs to be enabled

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.enable_systems<my_game::input, my_
↪game::render>();
    if (!result) {
        // Oh no, atleast one of the requested systems cannot be enabled.
    }
    return 0;
}
```

Return true if the list of systems has been enabled, false otherwise

See *enable_system*

template<typename **TSystem**>
bool **disable_system**()
This function disable a system.

Warning: If you deactivate a system, it will not be destroyed but simply ignored during the game loop.

Template Parameters

- `TSystem`: Represents the system that needs to be disabled

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.disable_system<my_game::input>();
    if (!result) {
        // Oh no the system_manager cannot disable this system.
    }
    return 0;
}
```

Return true if the the system has been disabled, false otherwise

template<typename ...**TSystems**>
bool **disable_systems**()
This function disable a list of systems.

Note: This function recursively calls the disable_system function

Template Parameters

- `TSystems`: Represents a list of systems that needs to be disabled

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};

    bool result = system_manager.disable_systems<my_game::input, my_
↪game::render>();
    if (!result) {
        // Oh no, atleast one of the requested systems cannot be disabled.
    }
    return 0;
}
```

Return true if the list of systems has been disabled, false otherwise

std::size_t **nb_systems**() **const**
This function returns the number of systems registered in the system manager.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    // added 2 systems here
    auto nb_systems = system_manager.nb_systems();
    if (nb_systems) {
        // Oh no, was expecting atleast 2 systems.
    }
    return 0;
}
```

Return number of systems

std::size_t **nb_systems**(*system_type sys_type*) **const**

This function returns the system number of a certain type to register in the system manager.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    // added 2 systems of update type here
    auto nb_systems = system_manager.nb_systems(system_type::pre_update);
    if (nb_systems) {
        // Oh no, was expecting atleast 2 systems of pre_update type.
    }
    return 0;
}
```

Parameters

- `sys_type`: represent the type of systems.

Return number of systems of a specific type.

template<typename **TSystem**, typename ...**TSystemArgs**>
TSystem &**create_system**(*TSystemArgs*&&... *args*)

This function allow you to create a system with the given argument. This function is a factory.

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    auto& foo_system = system_manager.create_system<my_system::foo>(); // you␣
↪can send argument of the foo constructor here.
    foo_system.update();
    return 0;
}
```

Template Parameters

- `TSystem`: represents the type of system to create
- `TSystemArgs`: represents the arguments needed to construct the system to create

Return Returns a reference to the created system

template<typename **TSystem**, typename ...**TSystemArgs**>
void **create_system_rt**(*TSystemArgs*&&... *args*)
TODO: Document it.

template<typename ...**TSystems**, typename ...**TArgs**>
auto **load_systems**(*TArgs*&&... *args*)
This function load a bunch os systems.

Note: This function recursively calls the create_system function

Template Parameters

- `TSystems`: represents a list of systems to be loaded

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/ecs/system.manager.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    antara::gaming::ecs::system_manager system_manager{entity_registry};
    auto&& [foo_system, bar_system] = system_manager.load_systems<my_
↪system::foo, my_system::bar>();
    foo_system.update();
    bar_system.update();
    return 0;
}
```

Return Tuple of systems loaded

See *create_system*

Private Types

using clock = std::chrono::steady_clock
Private typedefs.

sugar name for an chrono steady clock

using system_ptr = std::unique_ptr<base_system>
sugar name for a pointer to base_system

using system_array = std::vector<*system_ptr*>
sugar name for an array of system_ptr

using system_registry = std::array<*system_array*, *system_type*::size>
sugar name for a multidimensional array of system_array (pre_update, logic_update, post_update)

using systems_queue = std::queue<*system_ptr*>
sugar name for a queue of system pointer to add.

Private Functions

base_system &**add_system_**(*system_ptr* &&*system*, *system_type* *sys_type*)
Private member functions.

Private Members

entt::registry &**entity_registry_**
Private data members.

antara::gaming::ecs::system_type

enum antara::gaming::ecs::**system_type**
Enumeration that represents all possible system types in sdk gaming.

Values:

pre_update
Represents a pre_update system.

logic_update
Represents a logic system.

post_update
Represents a post_update system.

size
Represents the size of the enum.

using antara::gaming::ecs::**st_system_pre_update** = st::type<*system_type*, **struct** system_pre_update_tag>
strong_type relative to system_type::pre_update

using antara::gaming::ecs::**st_system_logic_update** = st::type<*system_type*, **struct** system_logic_update_tag>
strong_type relative to system_type::logic_update

using antara::gaming::ecs::**st_system_post_update** = st::type<*system_type*, **struct** system_post_update_tag>
strong_type relative to system_type::post_update

1.1.5 antara::gaming::event

antara::gaming::event::key_pressed

struct key_pressed
triggered when pressing a key on the keyboard.

Note: This class is automatically reflected for scripting systems such as lua, python.

Public Functions

key_pressed(input::key *key_*, bool *alt_*, bool *control_*, bool *shift_*, bool *system_*)
Constructors.

constructor with args

Note: Principal Constructor.

Parameters

- key_: represents the keyboard key currently pressed
- alt_: is true if the alt key on the keyboard is pressed
- control_: is true if the keyboard control key is pressed
- shift_: is true if the keyboard shift_ key is pressed
- system_: is true if the keyboard system key is pressed

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/event/key.pressed.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    dispatcher.trigger<key_pressed>(input::key::a, false, false, false,␣
↪false);
}
```

key_pressed()
default constructor (for scripting systems convenience)

Public Members

antara::gaming::input::key **key**
Fields.

key pressed

bool **alt** = {false}
is alt pressed at the same time.

bool **control** = {false}
is ctrl pressed at the same time.

bool **shift** = {false}
is shift pressed at the same time.

bool **system** = {false}
is system pressed at the same time.

antara::gaming::event::key_released

struct key_released
triggered when releasing a key on the keyboard.

Note: This class is automatically reflected for scripting systems such as lua, python.

Public Functions

key_released(input::key *key_*, bool *alt_*, bool *control_*, bool *shift_*, bool *system_*)
Constructors.

constructor with args

Note: Principal Constructor.

Parameters

- `key_`: represents the keyboard key currently released
- `alt_`: is true if the alt key on the keyboard is released
- `control_`: is true if the keyboard control key is released
- `shift_`: is true if the keyboard shift_ key is released
- `system_`: is true if the keyboard system key is released

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/event/key_released.hpp>

int main()
{
```

(continues on next page)

(continued from previous page)

```
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    dispatcher.trigger<key_released>(input::key::a, false, false, false,␣
↪false);
}
```

key_released()
default constructor (for scripting systems convenience)

Public Members

input::key **key**
Fields.

key released

bool **alt** = {false}
is alt released at the same time.

bool **control** = {false}
is ctrl released at the same time.

bool **shift** = {false}
is shift released at the same time.

bool **system** = {false}
is system released at the same time.

antara::gaming::event::quit_game

struct quit_game
Event that allows us to leave a game with a return value.

Note: This class is automatically reflected for scripting systems such as lua, python.

Public Functions

quit_game(int *return_value*)
Constructors.

constructor with args

Note: Principal Constructor.

Parameters

- `return_value`: The return value of the program when leaving the game

Example:

```
#include <entt/entity/registry.hpp>
#include <entt/dispatcher/dispatcher.hpp>
#include <antara/gaming/event/quit_game.hpp>

int main()
{
    entt::registry entity_registry;
    entt::dispatcher& dispatcher{registry.set<entt::dispatcher>()};
    dispatcher.trigger<quit_game>(0);
}
```

quit_game()
default constructor (for scripting systems convenience)

Public Members

int **return_value_**
Fields.

the return value of the program when leaving the game

Public Static Attributes

constexpr const event::invoker_dispatcher<*quit_game*, int> **invoker** = {}
Static fields.

1.1.6 antara::gaming::sfml

antara::gaming::sfml::audio_system

class audio_system : **public** antara::gaming::ecs::system<*audio_system*>
This class allows the customization and play of audio.

Public Functions

audio_system(entt::registry &*registry*)

Parameters

- `registry`: The entity_registry is provided to the system when it is created.

void **update**()
This function destroys and cleans up the sounds which are completed playing.

antara::gaming::sfml::component_sound

struct component_sound
This struct contains the sound and attributes of it such as volume.

Public Members

sf::Sound **sound**
This object is SFML's Sound instance which contains the sound data.

1.2 Antara Gaming Tutorials

1.2.1 Tutorial: Getting Started

1.2.2 antara-gaming-sdk

Antara Gaming Software Development Kit

Prerequisites

Below is the list of prerequisites to use the `antara-gaming-sdk` on your machine:

- CMake 3.14 minimum
- **clang-8** minimum (Windows/Linux/Osx) (clang and clang-cl both supported on Windows)
- **Optional** emscripten latest (Web)
- **Optional** Visual Studio 2019
- **Optional** Clang VS Toolset (installable through visual studio installer)

Getting Started

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

See deployment for notes on how to deploy the project on a live system.

Build

To build the project please follow the instructions below:

```
mkdir build ## bash or powershell
cd build ## bash or powershell

## Release or Debug are available
cmake -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_COMPILER=your_path_to_your_clang++ ../
↪#Linux / Osx
cmake -DCMAKE_BUILD_TYPE=Debug -G "Visual Studio 16 2019" -A x64 -T "ClangCl" -DCMAKE_
↪CXX_COMPILER="C:/Program Files/LLVM/bin/clang-cl.exe" ../ #Windows
```

(continues on next page)

(continued from previous page)

```
## We can even use Ninja for Windows / Linux / OSX
## On Windows you may want to open x64 Visual Studio Terminal Prompt for using Ninja
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_COMPILER=path_to_clang++ -DCMAKE_
↪C_COMPILER=path_to_clang ../

## Build (Debug / Release available)
cmake --build . --config Debug
```

There are also additional options with the CMake that allows to activate certain features of the SDK:

Table 1: CMake Options

Name	Description	How to enable it	Notes
USE_SFML_ANTARA_WRAPPER	Enable the SFML module of the SDK	-DUSE_SFML_ANTARA_WRAPPER=ON	Require SFML dependencies on Linux
USE_IMGUI_ANTARA_WRAPPER	Enable the IMGUI Support for the SDK	-DUSE_IMGUI_ANTARA_WRAPPER=ON	
ENABLE_BLOCKCHAIN_MODULES	Enable the Blockchain modules for the SDK (need additional dependencies)	-DENABLE_BLOCKCHAIN_MODULES=ON	
ANTARA_BUILD_DOCS	Enable the build of the documentation for the SDK	-DANTARA_BUILD_DOCS=ON	Require Sphinx And Doxygen
USE_LUA_ANTARA_WRAPPER	Enable the lua module for the SDK	-DUSE_LUA_ANTARA_WRAPPER=ON	
USE_ASAN	Enable the Address Sanitizer for the Unit tests of the SDK	-DUSE_ASAN=ON	Cannot be mixed with USE_TSAN and USE_UBSAN
USE_UBSAN	Enable the Undefined Behavior Sanitizer for the Unit tests of the SDK	-DUSE_UBSAN=ON	Cannot be mixed with USE_TSAN and USE_ASAN
USE_TSAN	Enable the Undefined Behavior Sanitizer for the Unit tests of the SDK	-DUSE_TSAN=ON	Cannot be mixed with USE_UBSAN and USE_ASAN
BUILD_WITH_APPIMAGE	Enable the AppImage auto-generation on Linux for bundle an executable builded with the SDK	-DBUILD_WITH_APPIMAGE=ON	Work only on Linux.
ENABLE_HTML_COMPILATION	Enable the HTML Compilation on Emscripten for an executable builded with the SDK	-DENABLE_HTML_COMPILATION=ON	Work only on Emscripten.
COVERAGE_CLION_IDE	Enable the Coverage inside CLion IDE.	-DCOVERAGE_CLION_IDE=ON	Work only with CLion IDE and Require ENABLE_COVERAGE
ANTARA_BUILD_EXAMPLES	Enable the example of the SDK.	-DANTARA_BUILD_EXAMPLES=ON	Some examples need mix of options such as USE_SFML_ANTARA_WRAPPER + ANTARA_BUILD_EXAMPLES
ANTARA_BUILD_UNIT_TESTS	Enable the unit tests of the SDK.	-DANTARA_BUILD_UNIT_TESTS=ON	Some examples need mix of options such as USE_LUA_ANTARA_WRAPPER + ANTARA_BUILD_UNIT_TESTS
USE_BOX2D_ANTARA_WRAPPER	Enable the Box2D modules of the SDK.	-DUSE_BOX2D_ANTARA_WRAPPER=ON	
ENABLE_COVERAGE	Enable the coverage macros for the SDK.	-DENABLE_COVERAGE=ON	

Installing

You do not need to install the gaming sdk, just use the CMake fetch_content command to use the project

Running the tests

Once you have compiled the sdk gaming with the option to enable unit tests.

They are located in the `bin/unit_tests` (Linux/Osx) or `bin/unit_tests/%CMAKE_BUILD_TYPE%` (Windows) folder

Deployment

construction

Built With

- doctest - The fastest feature-rich C++11/14/17/20 single-header testing framework for unit tests and TDD http://bit.ly/doctest-docs (MIT)
- doom-st - C++ implementation of strong types (MIT)
- doom-meta - Just a few metaprogramming utilities in C++ (MIT)
- loguru - A lightweight C++ logging library (Public Domain)
- fmt - A modern formatting library https://fmt.dev (MIT)
- nlohmann-json - JSON for Modern C++ https://nlohmann.github.io/json/ (MIT)
- EnTT - Gaming meets modern C++ - a fast and reliable entity-component system (ECS). (MIT)
- refl-cpp - A compile-time reflection library for modern C++ (MIT)
- range-v3 - Range library for C++14/17/20, basis for C++20's std::ranges (Boost Software License)
- expected - C++11/14/17 std::expected with functional-style extensions https://tl.tartanllama.xyz (CC0 1.0 Universal)
- (optional)ImGui - Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies (MIT)
- (optional)ImGui-SFML - ImGui binding for use with SFML (MIT)
- (optional)SFML - Simple and Fast Multimedia Library http://www.sfml-dev.org/
- (optional)reproc - Cross-platform (C99/C++11) process library (MIT)
- (optional)lua - The Lua repo, as seen by the Lua team. (MIT)
- (optional)sol2 - Sol3 (sol2 v3.0) - a C++ <-> Lua API wrapper with advanced features and top notch performance - is here, and it's great! Documentation: http://sol2.rtfd.io/ (MIT)
- (optional)restclient-cpp - C++ client for making HTTP/REST requests http://code.mrtazz.com/restclient-cpp/ (MIT)
- (optional)box2D - Box2D is a 2D physics engine for games http://box2d.org (ZLib)

Code of Conduct

Before any contribution please read our CODE OF CONDUCT.

Contributing

Please read CONTRIBUTING.md, contain the process for submitting pull requests to us.

Versioning

We use SemVer for versioning. For the versions available, see the tags on this repository.

Authors

- **Roman Sztergbaum** - *Co-Creator & Lead Dev* - Milerius
- **Tolga Ay** - *Co-Creator* - naezith

See also the list of contributors who participated in this project.

Contributors

Please read CONTRIBUTORS.md, contains the list of contributors.

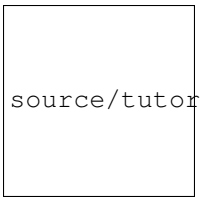
Acknowledgments

- Viktor Kirilov onqtam for the awesome `doctest` framework.
- Michele Caini skypjack for the awesome `EnTT` framework and his help.

Gallery

source/tutorials/docs/assets/gallery/wolf-ig2.png

source/tutorials/docs/assets/gallery/flappy.png



Badges

Apps	Badges
Github Actions CI (Windows/Osx/Linux)	
License	This work is licensed under a . **Our team is working on a more open license.**
LGTM (Security C++) Alerts	
LGTM (Security C++) Quality	
Issues	
Report CI (Linux/Osx/Windows/Emscripten)	
Coverage (Codecov)	
Docs	
HitCount	
Line Of Code	
Conventional Commit	

1.2.3 Tutorial: Quick And Dirty

If you have not read the *getting started* part yet, I invite you to do it now for the rest of this tutorial.

prerequisites

You'll need to have a basic `CMakeLists.txt` to able to compile your code:

```
##! Uncomment those lines if you use the gaming sdk as an external project
#if (${CMAKE_SYSTEM_NAME} STREQUAL "Linux")
#    set(LINUX TRUE)
#endif ()

#include(FetchContent)

#FetchContent_Declare(
#        antara-gaming-sdk
#        URL https://github.com/KomodoPlatform/antara-gaming-sdk/archive/master.zip
#)

#FetchContent_MakeAvailable(antara-gaming-sdk)
#init_apple_env()

add_executable(quick_and_dirty quick_and_dirty.cpp)
target_link_libraries(quick_and_dirty PUBLIC antara::world)
```

And below a cpp file with the primitives needed to launch your game:

```
/*************************************************************************
 * Copyright © 2013-2019 The Komodo Platform Developers.                 *
 *                                                                       *
 * See the AUTHORS, DEVELOPER-AGREEMENT and LICENSE files at             *
 * the top-level directory of this distribution for the individual copyright  *
 * holder information and the developer policies on copyright and licensing.  *
 *                                                                       *
 * Unless otherwise agreed in a custom licensing agreement, no part of the    *
 * Komodo Platform software, including this file may be copied, modified,     *
 * propagated or distributed except according to the terms contained in the   *
 * LICENSE file                                                          *
 *                                                                       *
 * Removal or modification of this copyright notice is prohibited.       *
 *                                                                       *
 *************************************************************************/

#include <iostream>
#include <antara/gaming/core/safe.refl.hpp>
#include <antara/gaming/world/world.app.hpp>

class example_system final : public antara::gaming::ecs::post_update_system<example_
↪system>
{
public:
    example_system(entt::registry& entity_registry) noexcept : system(entity_registry)
    {
        //! Here you can initialize your system, adding entities etc
    }

    void update() noexcept final
    {
        //! Your game logic here
        nb_iteration += 1;
        std::cout << "nb_iteration: " << nb_iteration << "\n";
        if (nb_iteration == 10ull) {
            std::cout << "Maximum iteration reached, leaving game now\n";
            this->dispatcher_.trigger<antara::gaming::event::quit_game>(0);
        }
    }

private:
    std::size_t nb_iteration{0ull};
};

REFL_AUTO(type(example_system));

class my_world_example : public antara::gaming::world::app
{
public:
    my_world_example() noexcept
    {
        this->system_manager_.create_system<example_system>(); //! Here we load our␣
↪system to use it.
    }
};

int main()
```

(continues on next page)

(continued from previous page)

```
{
    my_world_example world;
    return world.run();
}
```

1.2.4 Tutorial: Antara Gaming 2D Animations

1.2.5 Tutorial: Antara Gaming Systems

If you have not read the *getting started* part yet, I invite you to do it now for the rest of this tutorial.

If you have not read the ecs module documentation yet, I invite you to do it now to understand what we are doing in this tutorial series.

How to create your own system step by step ?

Setup

In this tutorial I will assume that you want to write a system for a project outside the gaming SDK. (An External Game Project)

Firstly we will need a **CMakeLists.txt**:

```
if (${CMAKE_SOURCE_DIR} STREQUAL ${CMAKE_BINARY_DIR})
    message(FATAL_ERROR "Prevented in-tree build. Please create a build directory␣
↪outside of the source code and call cmake from there")
endif ()

cmake_minimum_required(VERSION 3.14)

set(CMAKE_CXX_STANDARD 17)

project(my_game_project DESCRIPTION "my_game_description" LANGUAGES CXX)

if (NOT "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
    message(FATAL_ERROR "Only Clang is supported (minimum LLVM 8.0)")
endif ()

if (${CMAKE_SYSTEM_NAME} STREQUAL "Linux")
    set(LINUX TRUE)
endif ()

include(FetchContent)

FetchContent_Declare(
        antara-gaming-sdk
        URL https://github.com/KomodoPlatform/antara-gaming-sdk/archive/master.zip
)

FetchContent_MakeAvailable(antara-gaming-sdk)
init_apple_env()

add_executable(${PROJECT_NAME} my_example_system.cpp)
target_link_libraries(${PROJECT_NAME} PUBLIC antara::world)
```

As the `CMakeLists.txt` suggests we also need a **C++** files named `my_example_system.cpp` with the following contents:

```
#include <antara/gaming/world/world.app.hpp>

using namespace antara::gaming;

class my_world : public world::app
{
public:
    my_world() noexcept
    {

    }
};

int main()
{
    my_world world;
    return world.run();
}
```

And now we can successfully build the setup project that we just made:

> **Warning:** The project is build on OSX in the following video, if you want to build for your platform please refer to the *getting started* tutorial

Create a system

Let's create between the `using namespace` statement and the definition of the class `my_world` a system class who will be a `pre_update_system`

```
class pre_concrete_system final : public ecs::pre_update_system<pre_concrete_system>
{
public:
    //! Here the constructor can take other additional arguments but the first two␣
↪are mandatory
    pre_concrete_system(entt::registry &registry) noexcept : system(registry)
    {

    }

    void update() noexcept final
    {
        //! Empty for the moment
    }

    ~pre_concrete_system() noexcept final = default;
};
```

Now we can load this system into our world. Place yourself at the body of the constructor of the class `my_world`.

In order to load the system we will use the function `create_system` of the `system_manager` class.

```
my_world() noexcept
{
    //! Here we don't need to add any parameters for the constructor
    //! because the mandatory parameters are forwarded by default
    this->system_manager_.create_system<pre_concrete_system>();
}
```

Now, if you compile your program and start it, you will realize that you are in an infinite loop, that's simply mean your system is running inside the game loop.

But do not panic we have a way to stop our system thanks to the dispatcher.

We will make sure that after a number of iterations from our system, we will emit a `quit_game event` that will be catched by the world and stop the gaming loop.

To do this we will create a counter as a private field of our system and increment it each time the update function is called, arrived at 10 iterations we will emit an event to leave the game

```
class pre_concrete_system final : public antara::gaming::ecs::pre_update_system<pre_
↪concrete_system>
{
public:
    //! Here the constructor can take other additional arguments but the first two␣
↪are mandatory
    pre_concrete_system(entt::registry &registry) noexcept : system(registry)
    {

    }

    void update() noexcept final
    {
        std::cout << "nb_iteration: " << (++nb_iteration) << "\n";
        if (nb_iteration == 10u) {
            this->dispatcher_.trigger<antara::gaming::event::quit_game>(0);
        }
    }

    ~pre_concrete_system() noexcept final = default;
private:
    std::size_t nb_iteration{0u};
};

REFL_AUTO(type(pre_concrete_system)) //! This line is very important, it's give a␣
↪static reflection name function to your system, otherwise you will not compile.
```

Warning: Let's not forget the inclusion of iostream header at the top of the file.

Now, if you compile your program and start it you will quit your game after 10 iterations.

Below all the code of this tutorial:

```
#include <iostream>
#include <antara/gaming/world/world.app.hpp>

using namespace antara::gaming;
```

(continues on next page)

(continued from previous page)

```
class pre_concrete_system final : public ecs::pre_update_system<pre_concrete_system>
{
public:
    //! Here the constructor can take other additional arguments but the first two␣
↪are mandatory
    pre_concrete_system(entt::registry &registry) noexcept : system(registry)
    {

    }

    void update() noexcept final
    {
        std::cout << "nb_iteration: " << (++nb_iteration) << "\n";
        if (nb_iteration == 10u) {
            this->dispatcher_.trigger<antara::gaming::event::quit_game>(0);
        }
    }

    ~pre_concrete_system() noexcept final = default;
private:
    std::size_t nb_iteration{0u};
};

REFL_AUTO(type(pre_concrete_system))

class my_world : public world::app
{
public:
    my_world() noexcept
    {
        //! Here we don't need to add any parameters for the constructor
        //! because the mandatory parameters are forwarded by default
        this->system_manager_.create_system<pre_concrete_system>();
    }
};

int main()
{
    my_world world;
    return world.run();
}
```

Congratulations, you have managed to create your own system and add it to your game world !

1.2.6 Tutorial: Antara Gaming Events

1.2.7 Tutorial: How to do a Tic-Tac-Toe in less than 15 minutes with the gaming SDK ?

If you have not read the *getting started* part yet, Please read it before reading this one.

This tutorial is divided into multiple steps to make it easier to follow.

Step 1: Setup the executable and the window

First, create a folder called `tic-tac-toe` for your project, then create a text file and save it as `CMakeLists.txt` to create and compile the executable.

In this `CMakeLists.txt` file we will have: name of the project, creation of the executable, link with the SDK, C++ standard that will be used and extra modules that we want to use - in our case it will be the module antara::sfml provided by the **SDK**.

Below is the `CMakeLists.txt` file:

```
if (${CMAKE_SOURCE_DIR} STREQUAL ${CMAKE_BINARY_DIR})
    message(FATAL_ERROR "Prevented in-tree build. Please create a build directory␣
↪outside of the source code and call cmake from there")
endif ()

##! Minimum version of the CMake.
cmake_minimum_required(VERSION 3.14)

##! C++ Standard needed by the SDK is 17
set(CMAKE_CXX_STANDARD 17)

##! Our Project title, here tic-tac-toe.
project(tic-tac-toe DESCRIPTION "An awesome tic-tac-toe" LANGUAGES CXX)

##! The SDK need's clang as main compiler.
if (NOT "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
    if (NOT "${CMAKE_CXX_COMPILER_ID}" STREQUAL "AppleClang")
        message(FATAL_ERROR "Only Clang is supported (minimum LLVM 8.0)")
    endif()
endif ()

##! We will let know the SDK if our on Linux
if (${CMAKE_SYSTEM_NAME} STREQUAL "Linux")
    set(LINUX TRUE)
endif ()

##! We include the module from CMake for fetching dependencies
include(FetchContent)

##! We declare information about the dependance that we want to fetch.
FetchContent_Declare(
        antara-gaming-sdk
        URL https://github.com/KomodoPlatform/antara-gaming-sdk/archive/master.zip
)

##! We set extras modules from the SDK that we want to use, here we will use the SFML␣
↪module.
set(USE_SFML_ANTARA_WRAPPER ON)

##! We fetch our dependence
FetchContent_MakeAvailable(antara-gaming-sdk)

##! Calling this macros provided by the sdk will if you are on Apple init the␣
↪environment for this OS (std::filesystem).
init_apple_env()

##! We create the executable with the project name
```

(continues on next page)

(continued from previous page)

```
add_executable(${PROJECT_NAME} tic-tac-toe.cpp)

##! Setting output directory
set_target_properties(${PROJECT_NAME}
        PROPERTIES
        RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin/"
        )

##! We link the SDK modules that we want to use to our executable
target_link_libraries(${PROJECT_NAME} PUBLIC antara::world antara::sfml)

##! Move DLL
if (WIN32)
    ADD_CUSTOM_COMMAND(TARGET ${PROJECT_NAME} POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy_directory "${SFML_BINARY_DIR}/lib" "$
↪{CMAKE_BINARY_DIR}/bin/"
            COMMENT "copying dlls ..."
            $<TARGET_FILE_DIR:${PROJECT_NAME}>
            )

    ADD_CUSTOM_COMMAND(TARGET ${PROJECT_NAME} POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy "${SFML_SOURCE_DIR}/extlibs/bin/x64/
↪openal32.dll" "${CMAKE_BINARY_DIR}/bin/openal32.dll"
            COMMENT "copying dlls ..."
            $<TARGET_FILE_DIR:${PROJECT_NAME}>
            )
endif ()
```

Then we create our input file for the application and call it `tic-tac-toe.cpp`.

We add an empty main function:

```
int main()
{
    return 0;
}
```

If you did everything correctly so far, you should have the following tree:

```
./tic-tac-toe
├── CMakeLists.txt
└── tic-tac-toe.cpp
```

Before continuing the tutorial, make sure that you have installed the required dependencies and your program compiles with the build commands available in the tutorial *getting started*. If needed, help is available in the Komodo Discord

Now we need a world representing the world of our game, to do this we use the following header file: `#include <antara/gaming/world/world.app.hpp>`

And a basic structure that we name `tic_tac_toe_world`. It will inherit from `antara::gaming::world::app` class.

And use the namespace `antara::gaming` to make naming easier.

Finally, we declare our new object in the body of the main function and we replace the return value with the return value of our game returned by the `run` function of the `class world::app`.

It gives us the following result:

```
#include <antara/gaming/world/world.hpp>

using namespace antara::gaming;

struct tic_tac_toe_world : world::app
{
    //! Our game entry point
    tic_tac_toe_world() noexcept = default;
};

int main()
{
    tic_tac_toe_world game;
    return game.run();
}
```

If you compile now and run your executable, you have an infinite loop and nothing will happen.

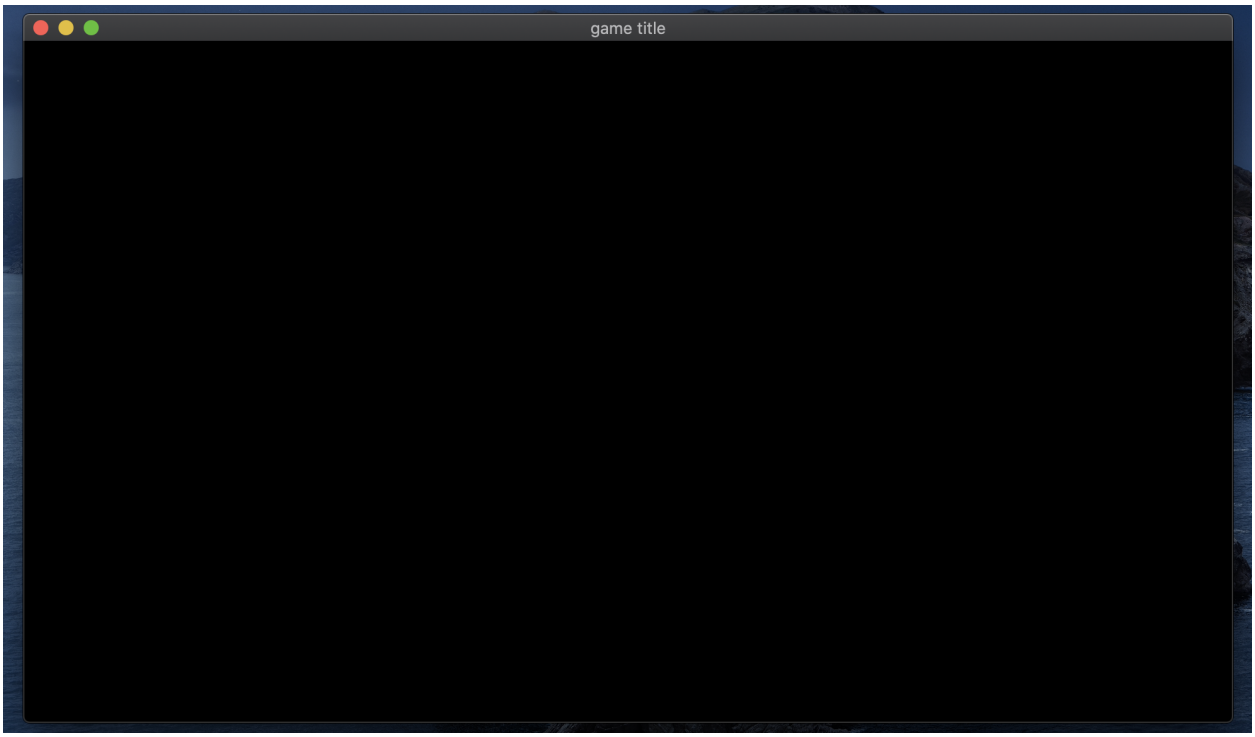
The last stage of this step one is to add the graphics side of the application, for that we will need two modules: `antara::gaming::sfml::graphic_system` and `antara::gaming::sfml::input::system` which have these following headers, respectively: `#include <antara/gaming/sfml/graphic.system.hpp>` and `#include <antara/gaming/sfml/input.system.hpp>`.

Now in the body of the constructor of our class `tic_tac_toe_world` we will load the graphic system, then we will initialize input system with the window coming from the loaded graphic system.

```
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>

//! For convenience
using namespace antara::gaming;

//! Our game world
struct tic_tac_toe_world : world::app
{
    //! Our game entry point
    tic_tac_toe_world() noexcept
    {
        //! Here we load our graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        //! Here we load our input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());
    }
};

int main()
{
    //! Here we declare our world
    tic_tac_toe_world game;

    //! Here we run the game
    return game.run();
}
```

If you compile and run now, you should see a black window open. You can close by pressing the close button of the

window:

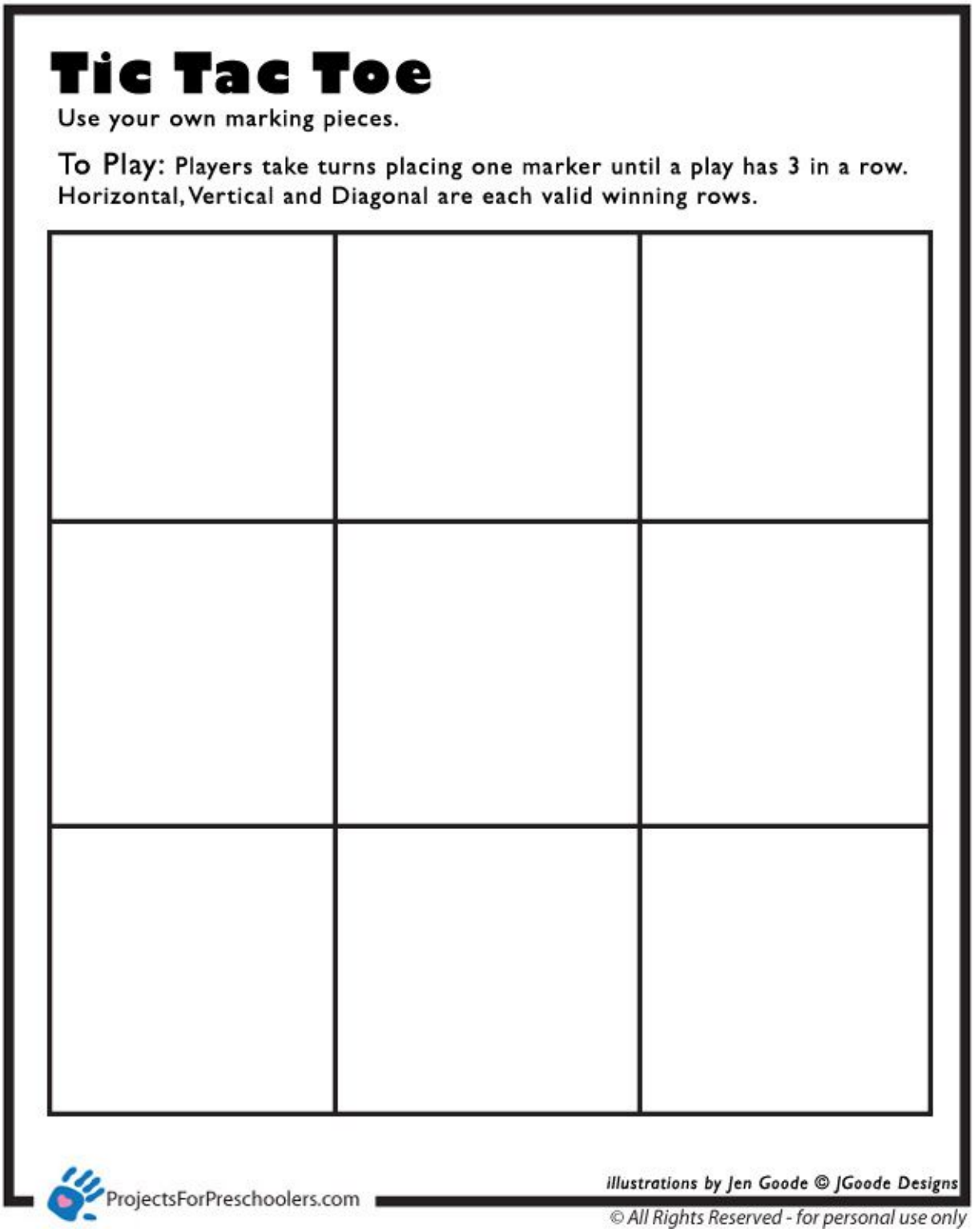


And now, the first step is over. We have a `CMakeLists.txt` to be able to compile our program into a basic executable which can create the game window.

Step 2: The Game Scene, The Grid, Game constants

For the second step, our goal is to draw the grid of Tic-Tac-Toe.

The grid will look like this:



To do this we will create a game scene using the scene manager. In order to do so we need to include the header file `#include <antara/gaming/scenes/scene.manager.hpp>` and load the scenes manager system into the system manager.

```
struct tic_tac_toe_world : world::app
{
    //! Our game entry point
    tic_tac_toe_world() noexcept
    {
        //! Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        //! Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        //! Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();
    }
};
```

Now we are going to create the `game_scene` class that inherits from the `base_scene` class. This class will be the entry point of our game scene.

The concrete class must override several functions such as update, scene_name, and the destructor. We will not use the update function because the Tic-Tac-Toe is not a game that needs an update for each frame, so we will leave the function empty. For the scene_name function we'll just return the name of the scene.

```
class game_scene final : public scenes::base_scene
{
public:
    game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
    {}

    //! This function will not be used, because Tic-Tac-Toe doesn't need an update␣
↪every frame.
    void update() noexcept final
    {}

    //! our scene name
    std::string scene_name() noexcept final
    {
        return "game_scene";
    }

    ~game_scene() noexcept final
    {}
private:
};
```

Now we are going to load our game scene into the `scene_manager` using the `change_scene` member function

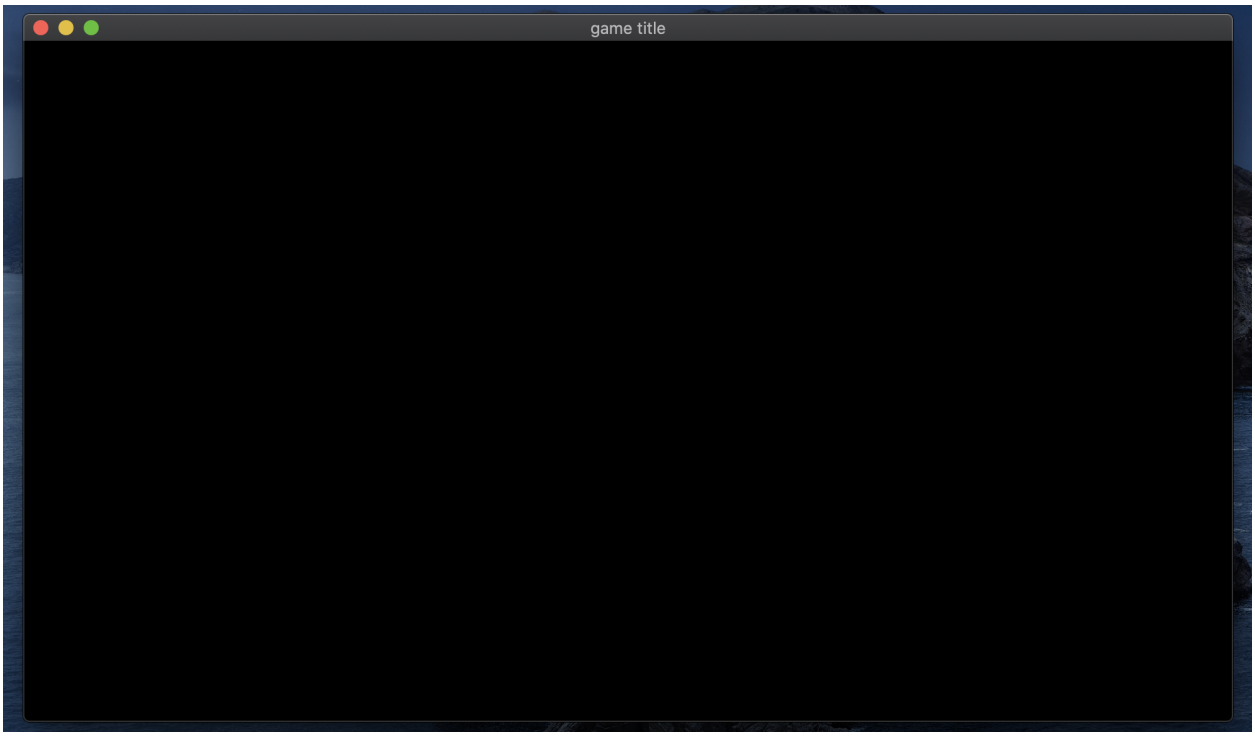
```
struct tic_tac_toe_world : world::app
{
    //! Our game entry point
    tic_tac_toe_world() noexcept
    {
        //! Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        //! Load the input system with the window from the graphical system
```

(continues on next page)

(continued from previous page)

```
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        //! Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        //! Change scene to game_scene
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};
```

If you compile now you should still see the black window from the previous step, but we are now in our game scene.



Note: The scene system is very handy to organize multiple screens of the game: **introduction scene**, **game scene**, **end-of-game scene**, etc.

Now we need several constants that are essential. For Tic-Tac-Toe they are: width and height of a cell, number of cells per line and thickness of the grid.

For the size of the cells we will use the current size of our canvas divided by the number of cells per line to obtain the size of a cell.

Now create a structure `tic_tac_toe_constants` that will contain these different information, then save it in the entity registry to be able to access from anywhere in the program.

```
struct tic_tac_toe_constants
{
    tic_tac_toe_constants(std::size_t nb_cells_per_axis_, std::size_t width_,␣
↪std::size_t height_) noexcept :
```

(continues on next page)

(continued from previous page)

```
            nb_cells_per_axis(nb_cells_per_axis_),
            cell_width(width_ / nb_cells_per_axis),
            cell_height(height_ / nb_cells_per_axis)
    {
    }

    const std::size_t nb_cells_per_axis;
    const std::size_t cell_width;
    const std::size_t cell_height;
    const float grid_thickness{20.0f};
};
```

In the constructor of the gaming scene:

```
game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
{
    //! Retrieve canvas information
    auto[canvas_width, canvas_height] = entity_registry_.ctx<graphics::canvas_2d>().
↪canvas.size.to<math::vec2u>();

    //! Set the constants that will be used in the program
    entity_registry_.set<tic_tac_toe_constants>(3ull, canvas_width, canvas_height);
}
```

Now we will go to the creation of our entity representing our grid, so we will add in private member of our game_scene class the `grid_entity_` field which is of type `entt::entity` which will have the initial value `entt::null`.

```
class game_scene final : public scenes::base_scene
{
public:
    game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
    {
        //! Retrieve canvas information
        auto[canvas_width, canvas_height] = entity_registry_.ctx<graphics::canvas_2d>
↪().canvas.size.to<math::vec2u>();

        //! Set the constants that will be used in the program
        entity_registry_.set<tic_tac_toe_constants>(3ull, canvas_width, canvas_
↪height);
    }

    //! This function won't be used, because Tic-Tac-Toe doesn't need to update every␣
↪frame.
    void update() noexcept final
    {}

    //! Return the scene name
    std::string scene_name() noexcept final
    {
        return "game_scene";
    }

    ~game_scene() noexcept final
    {}
private:
    //! The entity which represents the Tic-Tac-Toe grid
```

(continues on next page)

(continued from previous page)

```
    entt::entity grid_entity_{entt::null};
};
```

Then, we will have to initialize this entity. To do this we create an anonymous namespace with a function `create_grid` which returns an `entt::entity` and take in parameter `entity registry`.

```
//! Contains all functions which will be used for logic and factory
namespace
{
    //! Factory for creating a Tic-Tac-Toe grid
    entt::entity create_grid(entt::registry &registry) noexcept
    {
        return entt::null;
    }
}
```

Now, we call the function from the game scene constructor and we assign the return value to the field `grid_entity_`:

```
game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
{
    //! Retrieve canvas information
    auto[canvas_width, canvas_height] = entity_registry_.ctx<graphics::canvas_2d>().
↪canvas.size.to<math::vec2u>();

    //! Set the constants that will be used in the program
    entity_registry_.set<tic_tac_toe_constants>(3ull, canvas_width, canvas_height);

    //! Create the grid of the Tic-Tac-Toe
    grid_entity_ = create_grid(entity_registry_);
}
```

Only two things left to do now:

- code the logic of the `create_grid` function
- manage the destruction of the entities of our game scene when leaving the program

Let's start by coding the logic of the `create_grid` function.

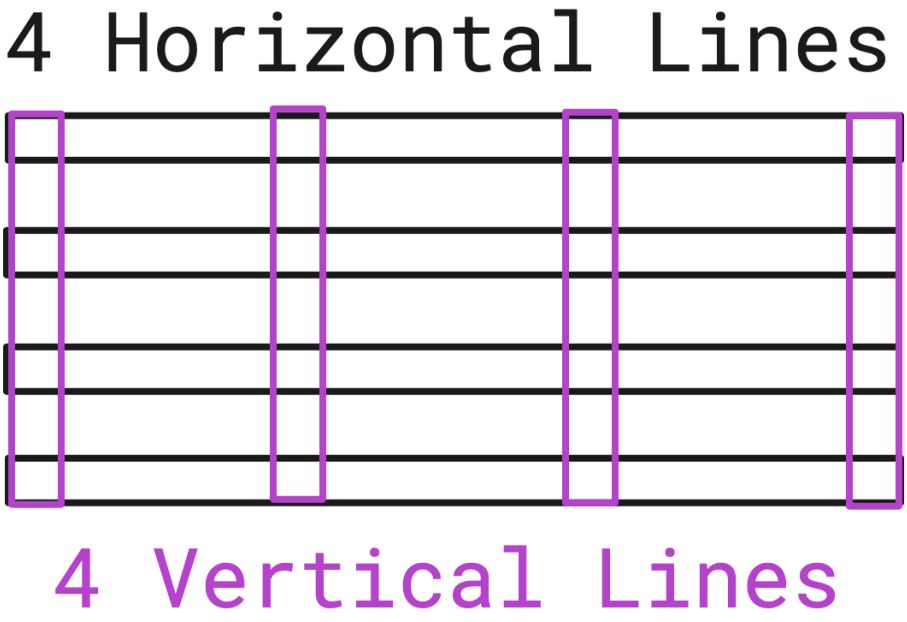
First we get the canvas size, because that will define the size of our grid.

```
//! Retrieve canvas information
auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.size;
```

Second, we create a new entity named grid.

```
//! Entity creation
auto grid_entity = registry.create();
```

A line is represented by joining two dots that we call vertices. Each vertex has a X position and a Y position. Connection of two vertices makes a line. Though that line thickness then would be `1 px`. `1 px` is not very visible if the image gets smaller because of scaling etc. So we want a thick line, like `20px`.



A thick line is basically a rectangle, right? For a rectangle, we need 4 vertices because of 4 corners. For a Tic-Tac-Toe grid, we need `4 vertical lines` (2 in middle and 2 at screen borders) and `4 horizontal lines`. That makes `8 lines`, and each line is `4 vertices`, so we need `8 * 4 = 32` vertices.

```
//! Our vertices
std::vector<geometry::vertex> lines{8 * 4};
```

We also need information about the grid, such as -

`nb_cells` = Number of cells in one axis (3 in this case).

`cell_width, cell_height` = Width and height of a cell.

`grid_thickness` = Thickness of the line.

We retrieve them from the defined constants:

```
//! Retrieve constants information
auto[nb_cells, cell_width, cell_height, grid_thickness] = registry.ctx<tic_tac_toe_
↪constants>();
```

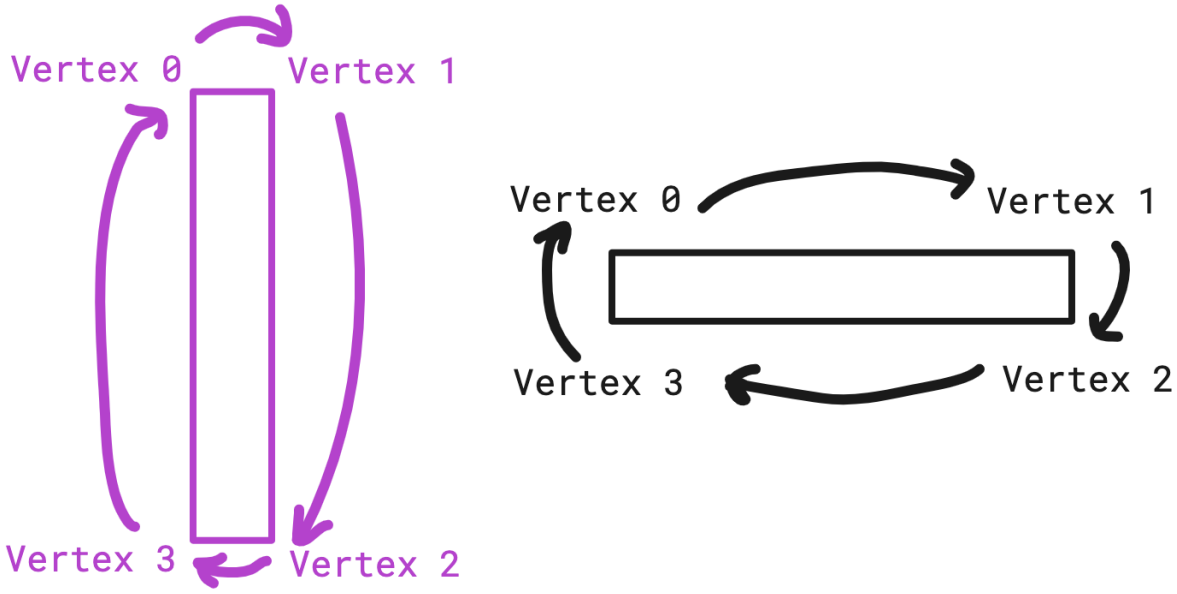
In calculations we will use half of the thickness for internal lines, and full thickness for the outer border. We can prepare for that with another constant for reuse and clarity.

```
const auto half_thickness = grid_thickness * 0.5f;
```

Our loop looks complicated but it actually isn't. At each loop we will define one vertical and one horizontal line. We have 4 lines in each axis so we need to loop 4 times, that is `nb_cells + 1`. Though we need to start from `0` because that will be used for the starting coordinate. We also have a variable `counter` which will count the vertex indexes, it will increase by `4 vertices * 2 lines = 8 vertices` at each iteration.

```
//! Our loop to create the grid
for (std::size_t counter = 0, i = 0; i <= nb_cells; ++i, counter += 4 * 2) {
```

Most important information is this: `Order of the vertices are always like this: Top Left, Top Right, Bottom Right, Bottom Left. So next neighbour is always the clockwise neighbour.`



Let's draw the vertical line first. Remember the order, we start with Top Left vertex. A vertical line is from top to bottom and `X` position will be same for top and bottom, but `Y` will change.

We calculate `X` first. `idx` is currently `0`, if we multiply that with `cell_width`, let's say `cell_width` is `300`, in `4` iterations these will be the values: `0, 300, 600, 900`.

Then we will do `- half_thickness` to offset it to a bit left because left and right vertices of a thick vertical line needs to be separate.

And the Y will be 0 because it's top of the screen. X axis grows from left to right, Y axis grows from top to down.

```
lines[counter + 0].pos = {idx * cell_width - half_thickness, 0.f};
```

Now, the Top Right vertex, it is same but it is `+ half_thickness` to offset in the opposite direction. Y is still `0` because it's top of the screen.

```
lines[counter + 1].pos = {idx * cell_width + half_thickness, 0.f};
```

Now, the Bottom Right vertex, `X` is same with Top Right, but `Y` is now canvas_height which is bottom of the screen.

```
lines[counter + 2].pos = {idx * cell_width + half_thickness, canvas_height};
```

Now, the last one, Bottom Left vertex, `Y` is same as Bottom Right, `X` is doing `- half_thickness` because it needs to be aligned to the left.

```
lines[counter + 3].pos = {idx * cell_width - half_thickness, canvas_height};
```

We completed the vertical line! Now we will do horizontal line. Again, always imagine these lines as rectangles. We will fill next 4 vertices now.

Starting at Top Left again, since line will be from left to right, `x` of left is `0`. And `Y` will change like `X` did in vertical line, 4 horizontal lines will have `Y` values as: `0, 300, 600, 900`.

Then we do `- half_thickness` to Y because Top Left needs to be at Top, we shift it a little bit to top to create the thickness.

```
lines[counter + 4].pos = {0, idx * cell_height - half_thickness};
```

Then Top Right vertex, is at far right side, `canvas_width` pixels away. And `Y` is same as Top Left.

```
lines[counter + 5].pos = {canvas_width, idx * cell_height - half_thickness};
```

Then Bottom Right vertex, `X` stays the same, and this time we add thickness to shift it to bottom, to create the thickness.

```
lines[counter + 6].pos = {canvas_width, idx * cell_height + half_thickness};
```

Then the last one, Bottom Left, is at far left, `0`. We shift `Y` to a bit bottom by adding thickness again.

```
lines[counter + 7].pos = {0, idx * cell_height + half_thickness};
```

Voilà ! Both vertical and horizontal lines are ready. Loop ends here.

After the loop, we turn these vertices to a `geometry::vertex_array` of quads, which are rectangles. And assign it to the `grid_entity`.

```
//! Assign the vertex array to the grid entity
registry.assign<geometry::vertex_array>(grid_entity, lines, geometry::vertex_geometry_
↪type::quads);
```

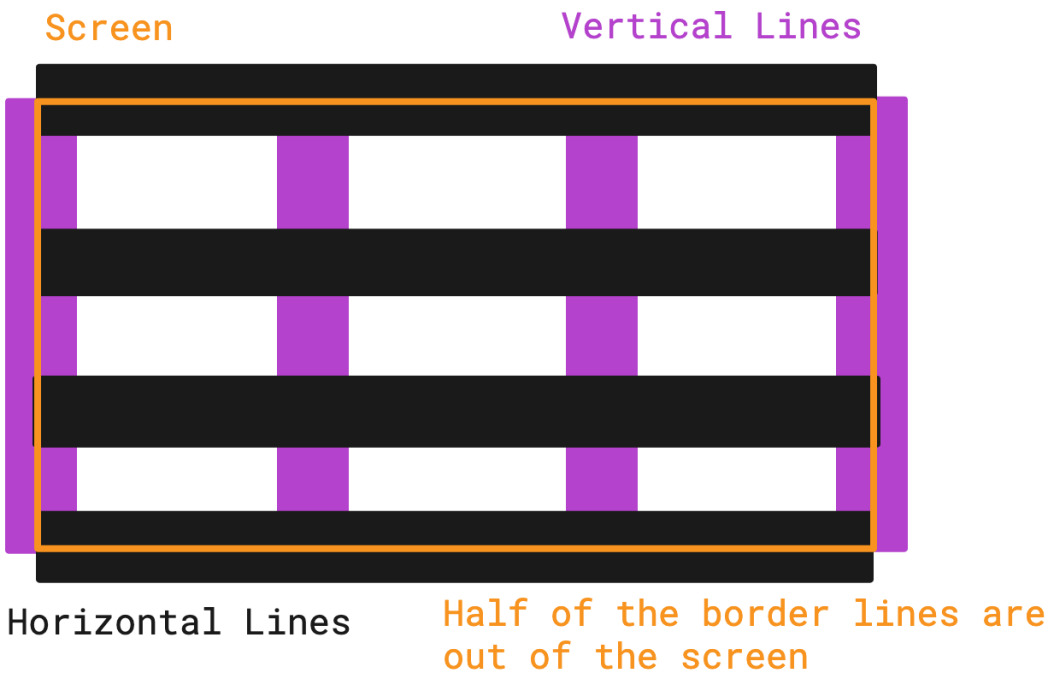
We tag the grid as `game_scene`

```
//! Assign the game_scene tag to the grid_entity (_hs means hashed_string)
registry.assign<entt::tag<"game_scene"_hs>>(grid_entity);
```

Set it to appear at `layer 0`, and return the prepared grid!

```
//! We want to draw the grid on the most deep layer, here 0.
registry.assign<graphics::layer<0>>(grid_entity);

//! We give back our fresh entity
return grid_entity;
```

This will work and look really good. Though maybe you realized, we always add and substract `half_thickness`. So the top border and left border of the screen are at coordinate `0`, so subtracting `half_thickness` will make half of it to appear out of the screen. Same with bottom border and right border, they are at `canvas_width` and `canvas_height` which are and of the screen. Adding `half_thickness` makes the half of it appear out of the screen again. If you are perfectionist, you don't want that to happen.



To solve this, we need to treat the first and last lines in a special way. We need to push top border a bit down, left border to a bit right, bottom border to a bit up and right border to a bit left to keep them fully inside the screen. We can call that shift `offset`, we go back to our loop and define it at start.

There is no offset by default, so we set them to `0`.

```
auto offset_x = 0.0f;
auto offset_y = 0.0f;
```

If it's the first ones, we add `half_thickness` to push them inside. And if it's last ones, we subtract `half_thickness` to pull them inside.

```
if (i == 0) {
    offset_x += half_thickness;
    offset_y += half_thickness;
}
else if (i == nb_cells) {
    offset_x -= half_thickness;
    offset_y -= half_thickness;
}
```

Now let's use the offsets we set.

For the vertical line, we use the `offset X` to push them left and right.

```
//! Vertical
lines[counter + 0].pos = {offset_x + idx * cell_width - half_thickness, 0.f};
lines[counter + 1].pos = {offset_x + idx * cell_width + half_thickness, 0.f};
lines[counter + 2].pos = {offset_x + idx * cell_width + half_thickness, canvas_height}
↪;
lines[counter + 3].pos = {offset_x + idx * cell_width - half_thickness, canvas_height}
↪;
```

For the horizontal line, we use the `offset Y` to push them up and down.

```
//! Horizontal
lines[counter + 4].pos = {offset_x + 0,            offset_y + idx * cell_height -␣
↪half_thickness};
lines[counter + 5].pos = {offset_x + canvas_width, offset_y + idx * cell_height -␣
↪half_thickness};
lines[counter + 6].pos = {offset_x + canvas_width, offset_y + idx * cell_height +␣
↪half_thickness};
lines[counter + 7].pos = {offset_x + 0,            offset_y + idx * cell_height +␣
↪half_thickness};
```

Now our grid must be looking absolutely perfect. You can edit `grid_thickness` constant to change the thickness of the lines.

Below the complete function:

```
//! Contains all the function that will be used for logic and factory
namespace
{
    //! Factory for creating a Tic-Tac-Toe grid
    entt::entity create_grid(entt::registry &registry) noexcept
    {
        //! Retrieve canvas information
        auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;

        //! Entity creation
        auto grid_entity = registry.create();

        //! Our vertices
        std::vector<geometry::vertex> lines{8 * 4};

        //! Retrieve constants information
        auto[nb_cells, cell_width, cell_height, grid_thickness] = registry.ctx<tic_
↪tac_toe_constants>();
        const auto half_thickness = grid_thickness * 0.5f;

        //! Our loop to create the grid
        for (std::size_t counter = 0, i = 0; i <= nb_cells; ++i, counter += 4 * 2) {

            //! To avoid narrowing conversion
            auto idx = static_cast<float>(i);

            //! First and last ones should be a bit inside, otherwise half of it is␣
↪out of the screen
            auto offset_x = 0.0f;
            auto offset_y = 0.0f;

            if (i == 0) {
                offset_x += half_thickness;
                offset_y += half_thickness;
            } else if (i == nb_cells) {
                offset_x -= half_thickness;
                offset_y -= half_thickness;
            }

            //! Prepare lines

            //! Vertical
```

(continues on next page)

(continued from previous page)

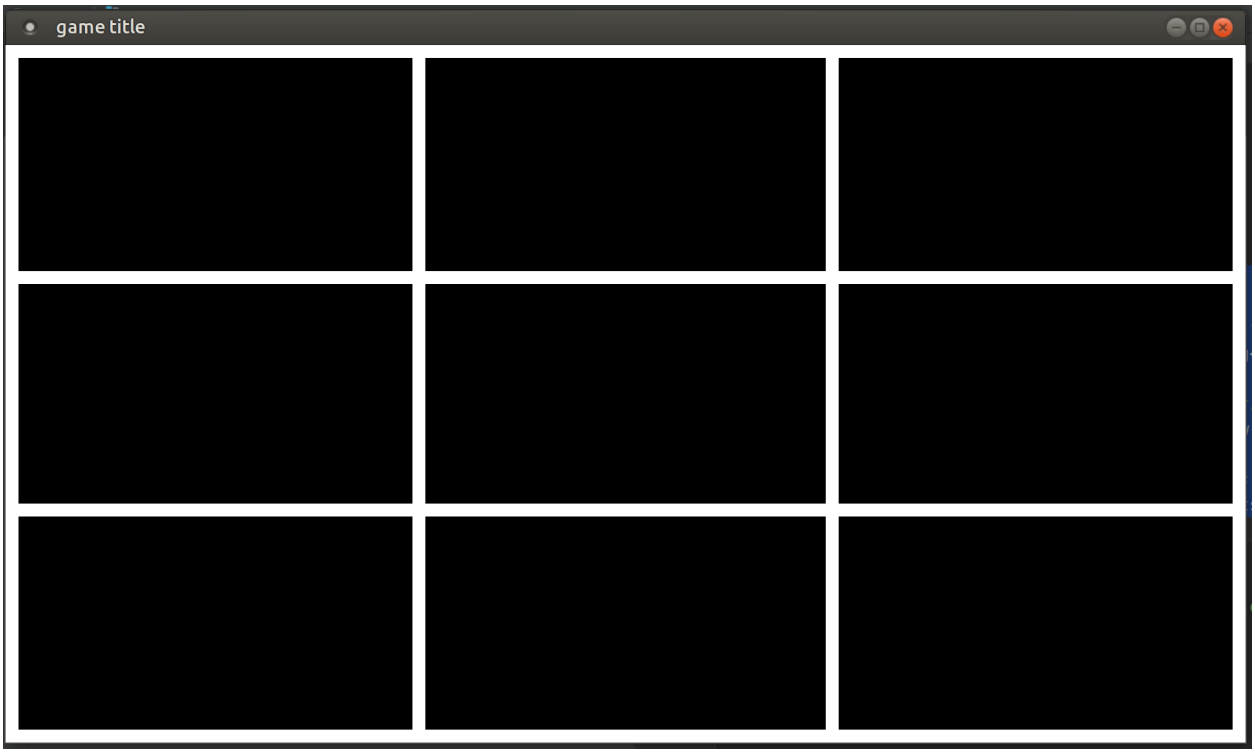
```
            lines[counter + 0].pos = {offset_x + idx * cell_width - half_thickness, 0.
↪f};
            lines[counter + 1].pos = {offset_x + idx * cell_width + half_thickness, 0.
↪f};
            lines[counter + 2].pos = {offset_x + idx * cell_width + half_thickness,␣
↪canvas_height};
            lines[counter + 3].pos = {offset_x + idx * cell_width - half_thickness,␣
↪canvas_height};

            //! Horizontal
            lines[counter + 4].pos = {offset_x + 0, offset_y + idx * cell_height -␣
↪half_thickness};
            lines[counter + 5].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height - half_thickness};
            lines[counter + 6].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height + half_thickness};
            lines[counter + 7].pos = {offset_x + 0, offset_y + idx * cell_height +␣
↪half_thickness};
        }

        //! Assign the vertex array to the grid entity
        registry.assign<geometry::vertex_array>(grid_entity, lines, geometry::vertex_
↪geometry_type::quads);

        //! Assign the game_scene tag to the grid_entity (_hs means hashed_string)
        registry.assign<entt::tag<"game_scene"_hs>>(grid_entity);

        //! We want to draw the grid on the most deep layer, here 0.
        registry.assign<graphics::layer<0>>(grid_entity);

        //! We give back our fresh entity
        return grid_entity;
    }
}
```

The last thing to do is the destruction of the entities in the destructor, we will need it at the time of reset the game.

In the destructor, we iterate over all the entities that have the tag of the game scene, and destroy each of them.

```
~game_scene() noexcept final
{
    //! Retrieve the collection of entities from the game scene
    auto view = entity_registry_.view<entt::tag<"game_scene"_hs>>();

    //! Iterate the collection and destroy each entities
    entity_registry_.destroy(view.begin(), view.end());

    //! Unset the Tic-Tac-Toe constants
    entity_registry_.unset<tic_tac_toe_constants>();
}
```

Now if you compile and run your program you should have the following result:



Here is the complete code of the second step:

```
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/graphics/component.layer.hpp>

//! For convenience
using namespace antara::gaming;

struct tic_tac_toe_constants
{
    tic_tac_toe_constants(std::size_t nb_cells_per_axis_, std::size_t width_,␣
↪std::size_t height_) noexcept :
            nb_cells_per_axis(nb_cells_per_axis_),
            cell_width(width_ / nb_cells_per_axis),
            cell_height(height_ / nb_cells_per_axis)
    {
    }

    const std::size_t nb_cells_per_axis;
    const std::size_t cell_width;
    const std::size_t cell_height;
    const float grid_thickness{20.0f};
};

//! Contains all the function that will be used for logic  and factory
namespace
{
```

(continues on next page)

(continued from previous page)

```
    //! Factory for creating a tic-tac-toe grid
    entt::entity create_grid(entt::registry &registry) noexcept
    {
        //! retrieve canvas information
        auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;

        //! entity creation
        auto grid_entity = registry.create();

        //! our vertices
        std::vector<geometry::vertex> lines{8 * 4};

        //! retrieve constants information
        auto[nb_cells, cell_width, cell_height, grid_thickness] = registry.ctx<tic_
↪tac_toe_constants>();
        const auto half_thickness = grid_thickness * 0.5f;

        //! our loop to create the grid
        for (std::size_t counter = 0, i = 0; i <= nb_cells; ++i, counter += 4 * 2) {

            //! to avoid narrowing conversion
            auto idx = static_cast<float>(i);

            //! first and last ones should be a bit inside, otherwise half of it is␣
↪out of the screen
            auto offset_x = 0.0f;
            auto offset_y = 0.0f;

            if (i == 0) {
                offset_x += half_thickness;
                offset_y += half_thickness;
            } else if (i == nb_cells) {
                offset_x -= half_thickness;
                offset_y -= half_thickness;
            }

            //! prepare lines

            //! vertical
            lines[counter + 0].pos = {offset_x + idx * cell_width - half_thickness, 0.
↪f};
            lines[counter + 1].pos = {offset_x + idx * cell_width + half_thickness, 0.
↪f};
            lines[counter + 2].pos = {offset_x + idx * cell_width + half_thickness,␣
↪canvas_height};
            lines[counter + 3].pos = {offset_x + idx * cell_width - half_thickness,␣
↪canvas_height};

            //! horizontal
            lines[counter + 4].pos = {offset_x + 0, offset_y + idx * cell_height -␣
↪half_thickness};
            lines[counter + 5].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height - half_thickness};
            lines[counter + 6].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height + half_thickness};
            lines[counter + 7].pos = {offset_x + 0, offset_y + idx * cell_height +␣
↪half_thickness};
```

(continues on next page)

(continued from previous page)

```
        }

        //! assign the vertex array to the grid entity
        registry.assign<geometry::vertex_array>(grid_entity, lines, geometry::vertex_
↪geometry_type::quads);

        //! assign the game_scene tag to the grid_entity (_hs means hashed_string)
        registry.assign<entt::tag<"game_scene"_hs>>(grid_entity);

        //! We want to draw the grid on the most deep layer, here 0.
        registry.assign<graphics::layer<0>>(grid_entity);

        //! we give back our fresh entity
        return grid_entity;
    }
}

class game_scene final : public scenes::base_scene
{
public:
    game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
    {
        //! Here we retrieve canvas information
        auto[canvas_width, canvas_height] = entity_registry_.ctx<graphics::canvas_2d>
↪().canvas.size.to<math::vec2u>();

        //! Here i set the constants that will be used in the program
        entity_registry_.set<tic_tac_toe_constants>(3ull, canvas_width, canvas_
↪height);

        //! Here i create the grid of the tic tac toe
        grid_entity_ = create_grid(entity_registry_);
    }

    //! This function will not be used, because tic tac toe doesn't need an update␣
↪every frame.
    void update() noexcept final
    {}

    //! our scene name
    std::string scene_name() noexcept final
    {
        return "game_scene";
    }

    ~game_scene() noexcept final
    {
        //! Here we retrieve the collection of entities from the game scene
        auto view = entity_registry_.view<entt::tag<"game_scene"_hs>>();

        //! Here we iterate the collection and destroy each entities
        entity_registry_.destroy(view.begin(), view.end());

        //! Here we unset the tic tac toe constants
        entity_registry_.unset<tic_tac_toe_constants>();
    }
```

(continues on next page)

(continued from previous page)

```
private:
    //! Our entity representing the tic-tac-toe grid
    entt::entity grid_entity_{entt::null};
};

//! Our game world
struct tic_tac_toe_world : world::app
{
    //! Our game entry point
    tic_tac_toe_world() noexcept
    {
        //! Here we load our graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        //! Here we load our input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        //! Here we load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        //! Here we change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};

int main()
{
    //! Here we declare our world
    tic_tac_toe_world game;

    //! Here we run the game
    return game.run();
}
```

Step 3: Create Board and X O, Game Logic

First we need `cell_state` to tell if the cell is empty, marked x or y. It's good to name these with `enum`.

```
enum cell_state
{
    empty,
    player_x = 1,
    player_y = 2
};
```

Now we will create the board which is `3x3`. We store value `3` in `nb_cells_per_axis` so we will use that. Board is a `vector` of 9 `cell_state` instances and all of them are marked as empty.

```
std::vector<cell_state> create_board(std::size_t nb_cells_per_axis)
{
    std::vector<cell_state> board(nb_cells_per_axis * nb_cells_per_axis, cell_
↪state::empty);
    return board;
```

(continues on next page)

(continued from previous page)

```
}
```

We also need to display X and O, let's prepare X first. Function will have `row` and `column` parameters.

```
void create_x(entt::registry &entity_registry, std::size_t row, std::size_t column)␣
↪noexcept
```

Now we get the constants like `nb_cells`, `cell_width`, `cell_height` and `grid_thickness`. Then create other helpful constants as `half_box_side`, `center_x`, `center_y` which is the center position of that specific cell.

```
auto[nb_cells, cell_width, cell_height, grid_thickness] = entity_registry.ctx<tic_tac_
↪toe_constants>();
const auto half_box_side = static_cast<float>(std::fmin(cell_width, cell_height) * 0.
↪25f);
const auto center_x = static_cast<float>(cell_width * 0.5 + column * cell_width);
const auto center_y = static_cast<float>(cell_height * 0.5 + row * cell_height);
```

We can make a X with two lines. Just like in Step 2, every line is a quad which has 4 vertices.

```
auto x_entity = entity_registry.create();
std::vector<geometry::vertex> lines{2 * 4};
```

X will have magenta color. We need to set every vertex color for it.

```
for (auto &&current_vertex: lines) current_vertex.pixel_color = graphics::magenta;
```

Again, just like in Step 2, we set the position of every single vertex. Order is Top Left, Top Right, Bottom Left, Bottom Right.

```
// Top-left to Bottom-right
lines[0].pos = {center_x - half_box_side - half_thickness, center_y - half_box_side};
lines[1].pos = {center_x - half_box_side + half_thickness, center_y - half_box_side};
lines[2].pos = {center_x + half_box_side + half_thickness, center_y + half_box_side};
lines[3].pos = {center_x + half_box_side - half_thickness, center_y + half_box_side};


// Top-right to Bottom-left
lines[4].pos = {center_x + half_box_side - half_thickness, center_y - half_box_side};
lines[5].pos = {center_x + half_box_side + half_thickness, center_y - half_box_side};
lines[6].pos = {center_x - half_box_side + half_thickness, center_y + half_box_side};
lines[7].pos = {center_x - half_box_side - half_thickness, center_y + half_box_side};
```

Create a `geometry::vertex_array` out of it.

```
entity_registry.assign<geometry::vertex_array>(x_entity, lines, geometry::vertex_
↪geometry_type::quads);
```

Assign this X entity as `player_x` and `game_scene`. Then set layer 1, because layer 0 is the background. X and O needs to render in front.

```
entity_registry.assign<entt::tag<"game_scene"_hs>>(x_entity);
entity_registry.assign<entt::tag<"player_x"_hs>>(x_entity);
entity_registry.assign<graphics::layer<1>>(x_entity);
```

Now X is complete and it's turn of O. First part is same as X.

```
void create_o(entt::registry &entity_registry, std::size_t row, std::size_t column)␣
↪noexcept
{
    auto constants = entity_registry.ctx<tic_tac_toe_constants>();
    const auto half_box_side = static_cast<float>(std::fmin(constants.cell_width,␣
↪constants.cell_height) * 0.25f);
    const auto center_x = static_cast<float>(constants.cell_width * 0.5 + column *␣
↪constants.cell_width);
    const auto center_y = static_cast<float>(constants.cell_height * 0.5 + row *␣
↪constants.cell_height);
```

To create the O, we create an entity first. Then assign it as a Circle, `geometry::circle`. After that, set the `fill_color` and `outline_color`. And finally set the entity position to center of the cell.

```
auto o_entity = geometry::blueprint_circle(entity_registry, half_box_side,␣
↪graphics::transparent,
        transform::position_2d(center_x, center_y),
        graphics::outline_color(constants.grid_thickness, graphics::cyan));
```

Last part is same as X, assigning to `game_scene` and set layer 1.

```
entity_registry.assign<entt::tag<"game_scene"_hs>>(o_entity);
entity_registry.assign<graphics::layer<1>>(o_entity);
```

Now the last part, we need to create a ecs::logic_update_system, we call it `tic_tac_toe_logic`.

```
class tic_tac_toe_logic final : public ecs::logic_update_system<tic_tac_toe_logic>
{
public:
    ~tic_tac_toe_logic() noexcept final = default;

    void update() noexcept final
    {}
```

As you see above, we don't do anything in update function because Tic-Tac-Toe is a passive game. We need to update only when mouse is clicked. But first let's define the `play_turn` function.

```
//! Game logic
void play_turn(std::size_t row, std::size_t column) noexcept
```

We turn row and column to index.

```
//! Retrieve constants
auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

//! Which cell is clicked ?
std::size_t index = row * constants.nb_cells_per_axis + column;
```

Then we make sure that index is inside the board and the clicked cell is empty. If so, we set the board as the current `player_turn_`. If it's X, we call `create_x` else we call `create_o` for that specific cell. Then change the turn to the other player.

```
    //! Cell is available ?
    if (index < board_.size() && board_[index] == cell_state::empty) {

        //! Change state of the cell to the current player
        board_[index] = static_cast<cell_state>(player_turn_);
```

(continues on next page)

(continued from previous page)

```
        //! Create x or o based on the current player
        player_turn_ == x ? create_x(entity_registry_, row, column) : create_o(entity_
↪registry_, row, column);

        //! Switch player
        player_turn_ = (player_turn_ == player::x) ? player::o : player::x;
    }
}
```

We call `play_turn` with the position of the mouse click.

```
void on_mouse_button_pressed(const event::mouse_button_pressed &evt) noexcept
 {
     if (current_game_state_ == running) {
         //! Retrieve game constants.
         auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

         //! Play one turn of the Tic-Tac-Toe
         play_turn(evt.y / constants.cell_height, evt.x / constants.cell_width);
     } else {
         //! Here we reset the game
     }
 }
```

Now we assign the `on_mouse_button_pressed` event to the mouse click in the constructor.

```
tic_tac_toe_logic(entt::registry &registry, entt::entity grid_entity, std::vector
↪<cell_state> board) noexcept
        : system(registry), grid_entity_(grid_entity), board_(std::move(board))
{
    //! stateless system
    this->disable();

    //! subscribe to mouse_button event
    this->dispatcher_.sink<event::mouse_button_pressed>().connect<&tic_tac_toe_
↪logic::on_mouse_button_pressed>(
            *this);
}
```

In the same class, we have enums `game_state` and `player` too.

```
//! Private enums
enum game_state
{
    running,
    player_x_won = 1,
    player_y_won = 2,
    tie,
    quit
};

enum player
{
    x = 1,
    o = 2
};
```

And the other member variables we used such as grid, state board, game state and player turn.

```
//! Private members variable
entt::entity grid_entity_{entt::null};
std::vector<cell_state> board_;
game_state current_game_state_{game_state::running};
player player_turn_{player::x};
```

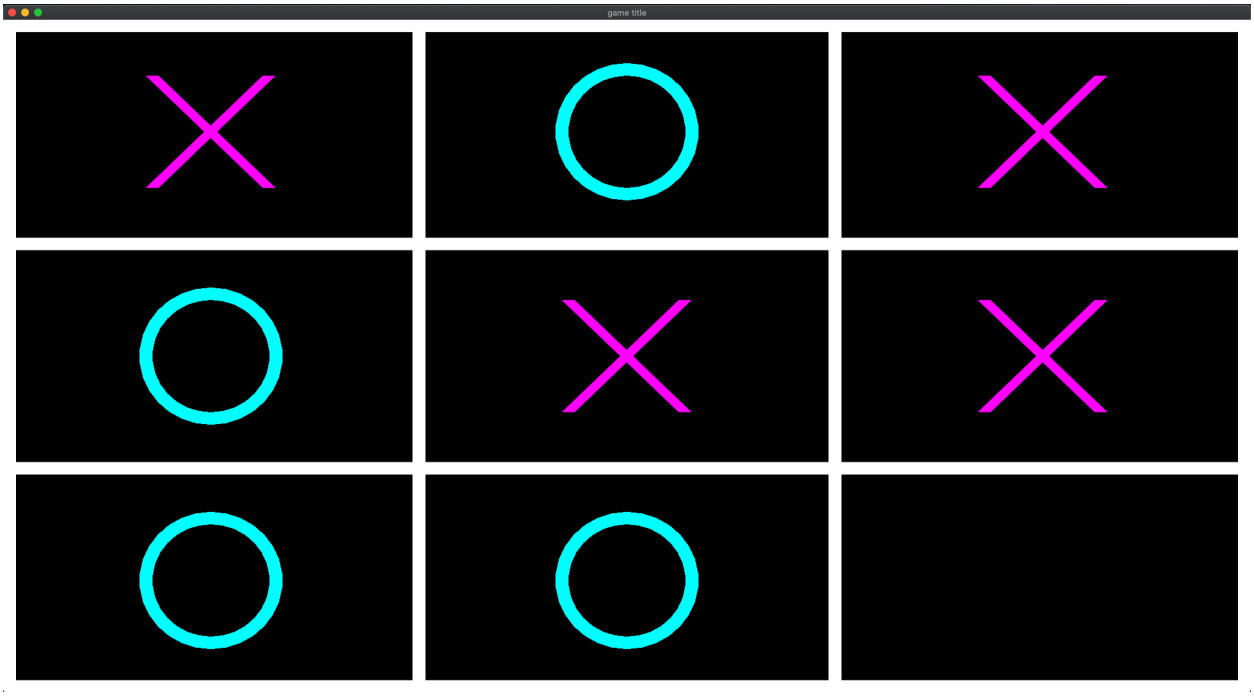
After the class definition, we give a name to our system, out of the class scope.

```
//! Give a name to our system
REFL_AUTO(type(tic_tac_toe_logic));
```

In the constructor of the game_scene we defined before, we create the board and the logic system.

```
//! Create the board of the tic tac toe
auto board = create_board(tictactoe_constants.nb_cells_per_axis);

//! Create our logic game system and give the fresh grid entity and the fresh board.
this->system_manager_.create_system<tic_tac_toe_logic>(grid_entity, board);
```

Game logic, board and XO, all of them are now complete.



Here is the complete code of the third step:

```
#include <vector>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/graphics/component.layer.hpp>
```

(continues on next page)

(continued from previous page)

```
//! For convenience
using namespace antara::gaming;

struct tic_tac_toe_constants
{
    tic_tac_toe_constants(std::size_t nb_cells_per_axis_, std::size_t width_,␣
↪std::size_t height_) noexcept :
            nb_cells_per_axis(nb_cells_per_axis_),
            cell_width(width_ / nb_cells_per_axis),
            cell_height(height_ / nb_cells_per_axis)
    {
    }

    const std::size_t nb_cells_per_axis;
    const std::size_t cell_width;
    const std::size_t cell_height;
    const float grid_thickness{20.0f};
};

enum cell_state
{
    empty,
    player_x = 1,
    player_y = 2
};

//! Contains all the function that will be used for logic  and factory
namespace
{
    //! Factory for creating a tic-tac-toe grid
    entt::entity create_grid(entt::registry &registry) noexcept
    {
        //! retrieve canvas information
        auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;

        //! entity creation
        auto grid_entity = registry.create();

        //! our vertices
        std::vector<geometry::vertex> lines{8 * 4};

        //! retrieve constants information
        auto[nb_cells, cell_width, cell_height, grid_thickness] = registry.ctx<tic_
↪tac_toe_constants>();
        const auto half_thickness = grid_thickness * 0.5f;

        //! our loop to create the grid
        for (std::size_t counter = 0, i = 0; i <= nb_cells; ++i, counter += 4 * 2) {

            //! to avoid narrowing conversion
            auto idx = static_cast<float>(i);

            //! first and last ones should be a bit inside, otherwise half of it is␣
↪out of the screen
            auto offset_x = 0.0f;
            auto offset_y = 0.0f;
```

(continues on next page)

(continued from previous page)

```
            if (i == 0) {
                offset_x += half_thickness;
                offset_y += half_thickness;
            } else if (i == nb_cells) {
                offset_x -= half_thickness;
                offset_y -= half_thickness;
            }

            //! prepare lines

            //! vertical
            lines[counter + 0].pos = {offset_x + idx * cell_width - half_thickness, 0.
↪f};
            lines[counter + 1].pos = {offset_x + idx * cell_width + half_thickness, 0.
↪f};
            lines[counter + 2].pos = {offset_x + idx * cell_width + half_thickness,␣
↪canvas_height};
            lines[counter + 3].pos = {offset_x + idx * cell_width - half_thickness,␣
↪canvas_height};

            //! horizontal
            lines[counter + 4].pos = {offset_x + 0, offset_y + idx * cell_height -␣
↪half_thickness};
            lines[counter + 5].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height - half_thickness};
            lines[counter + 6].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height + half_thickness};
            lines[counter + 7].pos = {offset_x + 0, offset_y + idx * cell_height +␣
↪half_thickness};
        }

        //! assign the vertex array to the grid entity
        registry.assign<geometry::vertex_array>(grid_entity, lines, geometry::vertex_
↪geometry_type::quads);

        //! assign the game_scene tag to the grid_entity (_hs means hashed_string)
        registry.assign<entt::tag<"game_scene"_hs>>(grid_entity);

        //! We want to draw the grid on the most deep layer, here 0.
        registry.assign<graphics::layer<0>>(grid_entity);

        //! we give back our fresh entity
        return grid_entity;
    }

    std::vector<cell_state> create_board(std::size_t nb_cells_per_axis)
    {
        std::vector<cell_state> board(nb_cells_per_axis * nb_cells_per_axis, cell_
↪state::empty);
        return board;
    }

    void create_x(entt::registry &entity_registry, std::size_t row, std::size_t␣
↪column) noexcept
    {
        auto[nb_cells, cell_width, cell_height, grid_thickness] = entity_registry.ctx
↪<tic_tac_toe_constants>();
```

(continues on next page)

(continued from previous page)

```
        const auto half_box_side = static_cast<float>(std::fmin(cell_width, cell_
↪height) * 0.25f);
        const auto center_x = static_cast<float>(cell_width * 0.5 + column * cell_
↪width);
        const auto center_y = static_cast<float>(cell_height * 0.5 + row * cell_
↪height);

        auto x_entity = entity_registry.create();
        std::vector<geometry::vertex> lines{2 * 4};

        for (auto &&current_vertex: lines) current_vertex.pixel_color =␣
↪graphics::magenta;

        const auto half_thickness = grid_thickness * 0.5f;

        // Top-left to Bottom-right
        lines[0].pos = {center_x - half_box_side - half_thickness, center_y - half_
↪box_side};
        lines[1].pos = {center_x - half_box_side + half_thickness, center_y - half_
↪box_side};
        lines[2].pos = {center_x + half_box_side + half_thickness, center_y + half_
↪box_side};
        lines[3].pos = {center_x + half_box_side - half_thickness, center_y + half_
↪box_side};


        // Top-right to Bottom-left
        lines[4].pos = {center_x + half_box_side - half_thickness, center_y - half_
↪box_side};
        lines[5].pos = {center_x + half_box_side + half_thickness, center_y - half_
↪box_side};
        lines[6].pos = {center_x - half_box_side + half_thickness, center_y + half_
↪box_side};
        lines[7].pos = {center_x - half_box_side - half_thickness, center_y + half_
↪box_side};

        entity_registry.assign<geometry::vertex_array>(x_entity, lines,␣
↪geometry::vertex_geometry_type::quads);
        entity_registry.assign<entt::tag<"game_scene"_hs>>(x_entity);
        entity_registry.assign<entt::tag<"player_x"_hs>>(x_entity);
        entity_registry.assign<graphics::layer<1>>(x_entity);
    }

    void create_o(entt::registry &entity_registry, std::size_t row, std::size_t␣
↪column) noexcept
    {
        auto constants = entity_registry.ctx<tic_tac_toe_constants>();
        const auto half_box_side = static_cast<float>(std::fmin(constants.cell_width,␣
↪constants.cell_height) * 0.25f);
        const auto center_x = static_cast<float>(constants.cell_width * 0.5 + column␣
↪* constants.cell_width);
        const auto center_y = static_cast<float>(constants.cell_height * 0.5 + row *␣
↪constants.cell_height);

        auto o_entity = geometry::blueprint_circle(entity_registry, half_box_side,␣
↪graphics::transparent,
                transform::position_2d(center_x, center_y),
```

(continues on next page)

(continued from previous page)

```
                graphics::outline_color(constants.grid_thickness, graphics::cyan));

        entity_registry.assign<entt::tag<"game_scene"_hs>>(o_entity);
        entity_registry.assign<graphics::layer<1>>(o_entity);
    }
}

class tic_tac_toe_logic final : public ecs::logic_update_system<tic_tac_toe_logic>
{
public:
    ~tic_tac_toe_logic() noexcept final = default;

    void update() noexcept final
    {}

    void on_mouse_button_pressed(const event::mouse_button_pressed &evt) noexcept
    {
        if (current_game_state_ == running) {
            //! Retrieve game constants.
            auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

            //! Play one turn of the Tic-Tac-Toe
            play_turn(evt.y / constants.cell_height, evt.x / constants.cell_width);
        } else {
            //! Reset the game
        }
    }

    tic_tac_toe_logic(entt::registry &registry, entt::entity grid_entity, std::vector
↪<cell_state> board) noexcept
            : system(registry), grid_entity_(grid_entity), board_(std::move(board))
    {
        //! stateless system
        this->disable();

        //! subscribe to mouse_button event
        this->dispatcher_.sink<event::mouse_button_pressed>().connect<&tic_tac_toe_
↪logic::on_mouse_button_pressed>(
                *this);
    }

private:
    //! Game logic
    void play_turn(std::size_t row, std::size_t column) noexcept
    {
        //! Retrieve constants
        auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

        //! Which cell is clicked ?
        std::size_t index = row * constants.nb_cells_per_axis + column;

        //! Cell is available ?
        if (index < board_.size() && board_[index] == cell_state::empty) {

            //! change state of the cell to the current player
            board_[index] = static_cast<cell_state>(player_turn_);
```

(continues on next page)

(continued from previous page)

```
            //! create x or o based on the current player
            player_turn_ == x ? create_x(entity_registry_, row, column) : create_
↪o(entity_registry_, row, column);

            //! switch player
            player_turn_ = (player_turn_ == player::x) ? player::o : player::x;
        }
    }

private:
    //! Private enums
    enum game_state
    {
        running,
        player_x_won = 1,
        player_y_won = 2,
        tie,
        quit
    };

    enum player
    {
        x = 1,
        o = 2
    };

private:
    //! Private members variable
    entt::entity grid_entity_{entt::null};
    std::vector<cell_state> board_;
    game_state current_game_state_{game_state::running};
    player player_turn_{player::x};
};

//! Give a name to our system
REFL_AUTO(type(tic_tac_toe_logic));

class game_scene final : public scenes::base_scene
{
public:
    game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
    {
        //! Retrieve canvas information
        auto[canvas_width, canvas_height] = entity_registry_.ctx<graphics::canvas_2d>
↪().canvas.size.to<math::vec2u>();

        //! Set the constants that will be used in the program
        auto &tictactoe_constants = entity_registry_.set<tic_tac_toe_constants>(3ull,␣
↪canvas_width, canvas_height);

        //! Create the grid of the tic tac toe
        auto grid_entity = create_grid(entity_registry_);

        //! Create the board of the tic tac toe
        auto board = create_board(tictactoe_constants.nb_cells_per_axis);

        //! Create our logic game system and give the fresh grid entity and the fresh␣
↪board.
```

(continues on next page)

(continued from previous page)

```
        this->system_manager_.create_system<tic_tac_toe_logic>(grid_entity, board);
    }

    //! This function will not be used, because tic tac toe doesn't need an update␣
↪every frame.
    void update() noexcept final
    {}


    //! our scene name
    std::string scene_name() noexcept final
    {
        return "game_scene";
    }

    ~game_scene() noexcept final
    {
        //! Retrieve the collection of entities from the game scene
        auto view = entity_registry_.view<entt::tag<"game_scene"_hs>>();

        //! Iterate the collection and destroy each entities
        entity_registry_.destroy(view.begin(), view.end());

        //! Unset the tic tac toe constants
        entity_registry_.unset<tic_tac_toe_constants>();
    }

private:
    ecs::system_manager system_manager_{entity_registry_};
};

//! Our game world
struct tic_tac_toe_world : world::app
{
    //! Our game entry point
    tic_tac_toe_world() noexcept
    {
        //! Load our graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        //! Load our input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        //! Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        //! Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};

int main()
{
    //! Declare our world
    tic_tac_toe_world game;
```

(continues on next page)

(continued from previous page)

```
    //! Run the game
    return game.run();
}
```

Step 4: Win, Tie conditions and reset game

At this final step, for this program to become a real game, we need to add win, tie conditions and the reset game functionality.

Let's create a blank `reset_event`.

```
struct reset_event
{

};
```

And define the reset event callback, it will call the destructor and call the constructor again.

```
//! Callback
void on_reset_event(const reset_event &) noexcept
{
    entt::registry &registry = this->entity_registry_;
    this->~game_scene();
    new(this) game_scene(registry);
}
```

We subscribe to this reset event in `game_scene` constructor.

```
class game_scene final : public scenes::base_scene
{
public:
    game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
    {
        //! Subscribe to reset event
        this->dispatcher_.sink<reset_event>().connect<&game_scene::on_reset_event>
↪(*this);
```

And we remove this event in the class destructor.

```
~tic_tac_toe_logic() noexcept final
{
    this->dispatcher_.sink<event::mouse_button_pressed>().disconnect(*this);
}
```

Then we trigger this when mouse button is pressed and game state isn't `running`.

```
this->dispatcher_.trigger<reset_event>();

---------

void on_mouse_button_pressed(const event::mouse_button_pressed &evt) noexcept
{
    if (current_game_state_ == running) {
        //! Retrieve game constants.
```

(continues on next page)

(continued from previous page)

```
        auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

        //! Play one turn of the Tic-Tac-Toe
        play_turn(evt.y / constants.cell_height, evt.x / constants.cell_width);
    } else {
        //! Reset the game
        this->dispatcher_.trigger<reset_event>();
    }
}
```

We nicely set-up the reset game functionality.

Now we need to define the checks for win and tie conditions. Let's start with the winning condition. It's enough to check only the current player's win situation. In this function we check every cell and count how many of them are marked as current player, separately in two variables `row_count` and `column_count`. If any of these two reached to `nb_cells` which is `3`, it's a win, we return true. Then we count both diagonal lines and check the same thing. If none of these match 3, it's not a win yet, returning false.

```
[[nodiscard]] bool did_current_player_win_the_game() const noexcept
{
    std::size_t row_count{0u}, column_count{0u}, diag1_count{0u}, diag2_count{0u};
    auto[nb_cells, cell_width, cell_height, _] = entity_registry_.ctx<tic_tac_toe_
↪constants>();
    for (std::size_t i = 0; i < nb_cells; ++i) {
        for (std::size_t j = 0; j < nb_cells; ++j) {
            //! Check rows
            if (board_[i * nb_cells + j] == static_cast<cell_state>(player_turn_))
                row_count++;

            //! Check columns
            if (board_[j * nb_cells + i] == static_cast<cell_state>(player_turn_))
                column_count++;
        }

        //! Check condition
        if (row_count >= nb_cells || column_count >= nb_cells) {
            return true;
        }

        //! Reset rows and columns
        row_count = 0u, column_count = 0u;

        //! Diag1 count
        if (board_[i * nb_cells + i] == static_cast<cell_state>(player_turn_))
            diag1_count++;

        //! Second diag count
        if (board_[i * nb_cells + nb_cells - i - 1] == static_cast<cell_state>(player_
↪turn_))
            diag2_count++;
    }

    //! Condition
    return diag1_count >= nb_cells || diag2_count >= nb_cells;
}
```

So we will check the winning condition first. If it's not a win, and if all cells are filled, then it's a tie. So tie

implementation is really easy.

```
[[nodiscard]] bool is_tie() const noexcept
{
    return std::count(begin(board_), end(board_), cell_state::empty) == 0;
}
```

Now we will use these two condition check functions in a bigger function which will be called later.

```
void check_winning_condition() noexcept
```

Inside it, we define a functor `make_screen` inside, which sets the color of the grid.

```
auto make_screen = [this](graphics::color clr_winner,
                          entt::entity entity) {
    auto &array_cmp = this->entity_registry_.get<geometry::vertex_array>(entity);
    for (auto &v : array_cmp.vertices) v.pixel_color = clr_winner;
    entity_registry_.replace<geometry::vertex_array>(entity, array_cmp.vertices,␣
↪array_cmp.geometry_type);
};
```

Using this functor, we make another one `make_player_win_screen` which gives the winner's color as an argument.

```
auto make_player_win_screen = [this, make_screen](entt::entity entity) {
    auto winning_color = player_turn_ == player::x ? graphics::magenta :␣
↪graphics::cyan;
    make_screen(winning_color, entity);
};
```

Same way, another one which feeds another color when it's tie.

```
auto make_tie_screen = [make_screen](entt::entity entity) {
    make_screen(graphics::yellow, entity);
};
```

Now let's use these two. First, we need to check if current player won the game, if not, we check if it isn a tie. Depending on that, we set the game state and call the proper functor we defined earlier.

```
if (did_current_player_win_the_game()) {
    current_game_state_ = static_cast<game_state>(player_turn_);
    make_player_win_screen(grid_entity_);
} else if (is_tie()) {
    current_game_state_ = game_state::tie;
    make_tie_screen(grid_entity_);
}
```

This function `check_winning_condition` looks like this when we sum it up:

```
void check_winning_condition() noexcept
{
    auto make_screen = [this](graphics::color clr_winner,
                              entt::entity entity) {
        auto &array_cmp = this->entity_registry_.get<geometry::vertex_array>(entity);
        for (auto &v : array_cmp.vertices) v.pixel_color = clr_winner;
        entity_registry_.replace<geometry::vertex_array>(entity, array_cmp.vertices,␣
↪array_cmp.geometry_type);
    };
```

(continues on next page)

(continued from previous page)

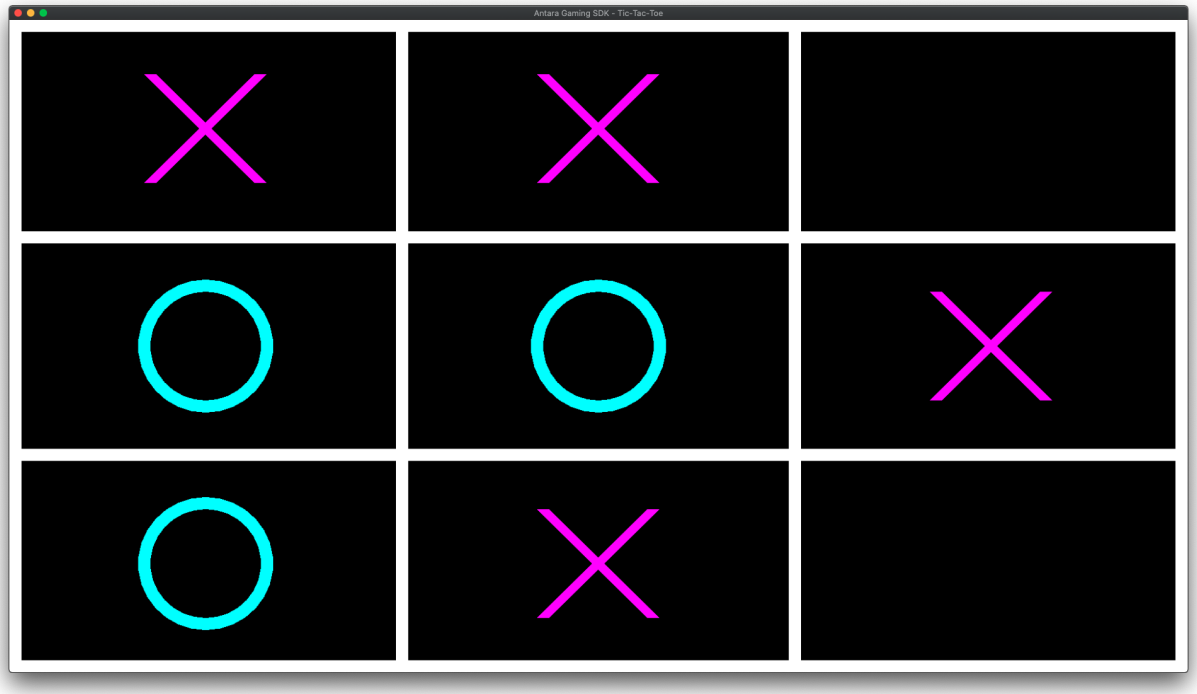
```
    auto make_player_win_screen = [this, make_screen](entt::entity entity) {
        auto winning_color = player_turn_ == player::x ? graphics::magenta :␣
↪graphics::cyan;
        make_screen(winning_color, entity);
    };

    auto make_tie_screen = [make_screen](entt::entity entity) {
        make_screen(graphics::yellow, entity);
    };

    if (did_current_player_win_the_game()) {
        current_game_state_ = static_cast<game_state>(player_turn_);
        make_player_win_screen(grid_entity_);
    } else if (is_tie()) {
        current_game_state_ = game_state::tie;
        make_tie_screen(grid_entity_);
    }
}
```

And finally we call this function in the end of `play_turn`.

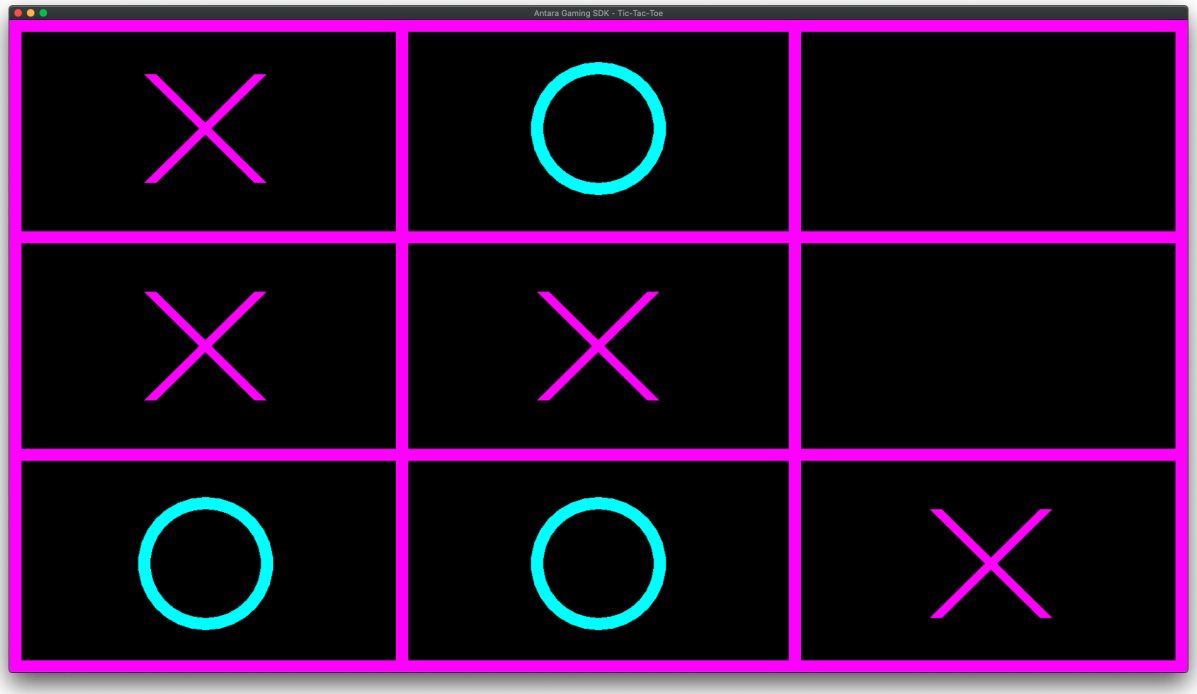
```
void play_turn(std::size_t row, std::size_t column) noexcept
{
    //! Retrieve constants
    auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

    //! Which cell is clicked ?
    std::size_t index = row * constants.nb_cells_per_axis + column;

    //! Cell is available ?
    if (index < board_.size() && board_[index] == cell_state::empty) {

        //! Change state of the cell to the current player
        board_[index] = static_cast<cell_state>(player_turn_);

        //! Create x or o based on the current player
        player_turn_ == x ? create_x(entity_registry_, row, column) : create_o(entity_
↪registry_, row, column);

        //! Check winning condition
        check_winning_condition();

        //! Switch player
        player_turn_ = (player_turn_ == player::x) ? player::o : player::x;
    }
}
```

That's it! Here are the screenshots:

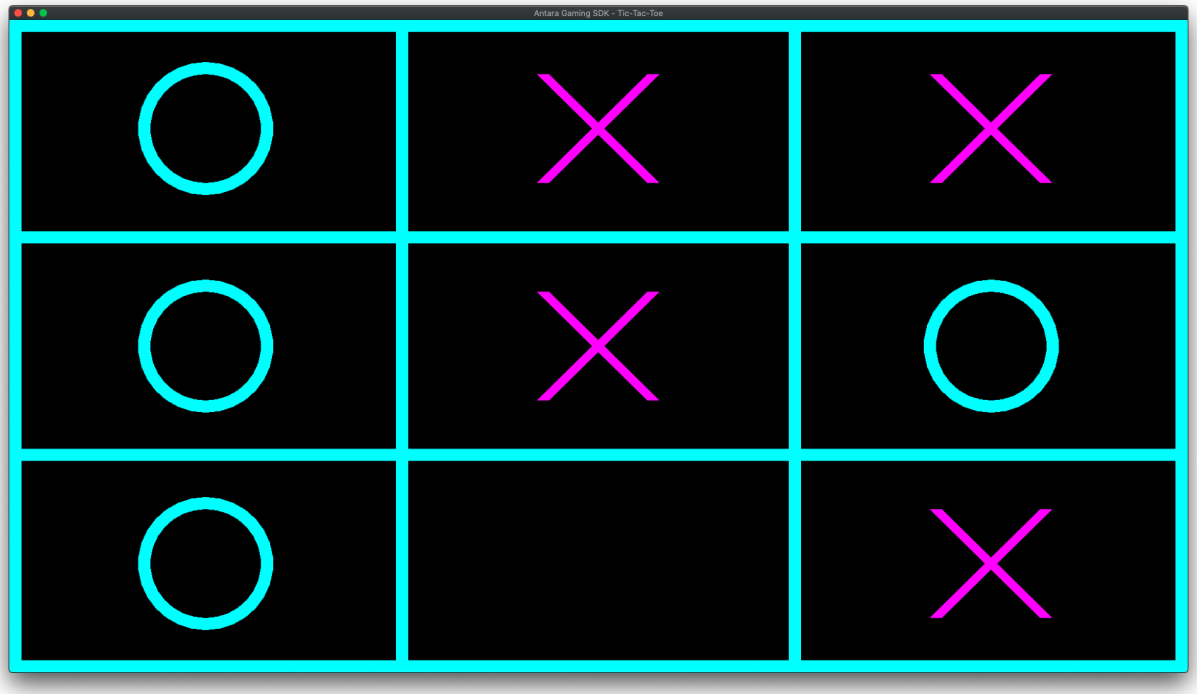
During the match:



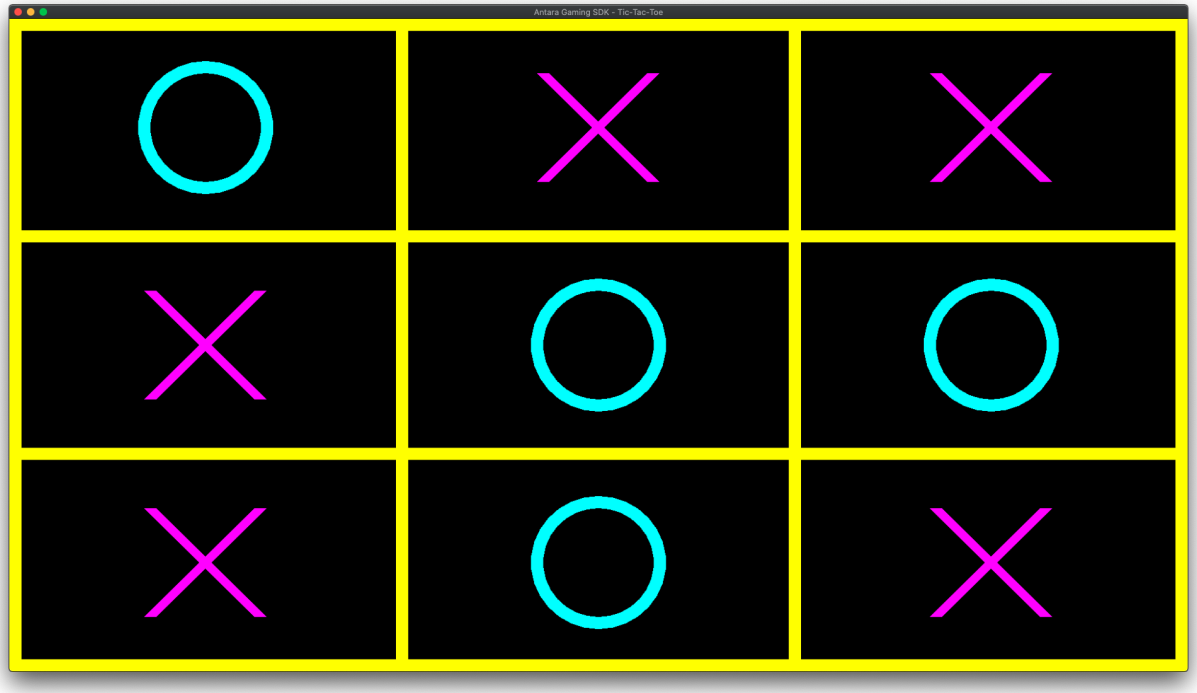
When Player X wins:



When Player O wins:



When it's a tie!



Here is the complete code of the last step:

```
#include <vector>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/graphics/component.layer.hpp>

//! For convenience
using namespace antara::gaming;

struct tic_tac_toe_constants
{
    tic_tac_toe_constants(std::size_t nb_cells_per_axis_, std::size_t width_,␣
↪std::size_t height_) noexcept :
            nb_cells_per_axis(nb_cells_per_axis_),
            cell_width(width_ / nb_cells_per_axis),
            cell_height(height_ / nb_cells_per_axis)
    {
    }

    const std::size_t nb_cells_per_axis;
    const std::size_t cell_width;
    const std::size_t cell_height;
    const float grid_thickness{20.0f};
};

enum cell_state
{
    empty,
    player_x = 1,
    player_y = 2
};

struct reset_event
{

};

//! Contains all the function that will be used for logic  and factory
namespace
{
    //! Factory for creating a tic-tac-toe grid
    entt::entity create_grid(entt::registry &registry) noexcept
    {
        //! Retrieve canvas information
        auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;

        //! Entity creation
        auto grid_entity = registry.create();

        //! Our vertices
        std::vector<geometry::vertex> lines{8 * 4};

        //! Retrieve constants information
        auto[nb_cells, cell_width, cell_height, grid_thickness] = registry.ctx<tic_
↪tac_toe_constants>();
        const auto half_thickness = grid_thickness * 0.5f;
```

(continues on next page)

(continued from previous page)

```
        //! Our loop to create the grid
        for (std::size_t counter = 0, i = 0; i <= nb_cells; ++i, counter += 4 * 2) {

            //! To avoid narrowing conversion
            auto idx = static_cast<float>(i);

            //! First and last ones should be a bit inside, otherwise half of it is␣
↪out of the screen
            auto offset_x = 0.0f;
            auto offset_y = 0.0f;

            if (i == 0) {
                offset_x += half_thickness;
                offset_y += half_thickness;
            } else if (i == nb_cells) {
                offset_x -= half_thickness;
                offset_y -= half_thickness;
            }

            //! Prepare lines

            //! Vertical
            lines[counter + 0].pos = {offset_x + idx * cell_width - half_thickness, 0.
↪f};
            lines[counter + 1].pos = {offset_x + idx * cell_width + half_thickness, 0.
↪f};
            lines[counter + 2].pos = {offset_x + idx * cell_width + half_thickness,␣
↪canvas_height};
            lines[counter + 3].pos = {offset_x + idx * cell_width - half_thickness,␣
↪canvas_height};

            //! Horizontal
            lines[counter + 4].pos = {offset_x + 0, offset_y + idx * cell_height -␣
↪half_thickness};
            lines[counter + 5].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height - half_thickness};
            lines[counter + 6].pos = {offset_x + canvas_width, offset_y + idx * cell_
↪height + half_thickness};
            lines[counter + 7].pos = {offset_x + 0, offset_y + idx * cell_height +␣
↪half_thickness};
        }

        //! Assign the vertex array to the grid entity
        registry.assign<geometry::vertex_array>(grid_entity, lines, geometry::vertex_
↪geometry_type::quads);

        //! Assign the game_scene tag to the grid_entity (_hs means hashed_string)
        registry.assign<entt::tag<"game_scene"_hs>>(grid_entity);

        //! We want to draw the grid on the most deep layer, here 0.
        registry.assign<graphics::layer<0>>(grid_entity);

        //! We give back our fresh entity
        return grid_entity;
    }
```

(continues on next page)

(continued from previous page)

```
    std::vector<cell_state> create_board(std::size_t nb_cells_per_axis)
    {
        std::vector<cell_state> board(nb_cells_per_axis * nb_cells_per_axis, cell_
↪state::empty);
        return board;
    }

    void create_x(entt::registry &entity_registry, std::size_t row, std::size_t␣
↪column) noexcept
    {
        auto[_, cell_width, cell_height, grid_thickness] = entity_registry.ctx<tic_
↪tac_toe_constants>();
        const auto half_box_side = static_cast<float>(std::fmin(cell_width, cell_
↪height) * 0.25f);
        const auto center_x = static_cast<float>(cell_width * 0.5 + column * cell_
↪width);
        const auto center_y = static_cast<float>(cell_height * 0.5 + row * cell_
↪height);

        auto x_entity = entity_registry.create();
        std::vector<geometry::vertex> lines{2 * 4};

        for (auto &&current_vertex: lines) current_vertex.pixel_color =␣
↪graphics::magenta;

        const auto half_thickness = grid_thickness * 0.5f;

        // Top-left to Bottom-right
        lines[0].pos = {center_x - half_box_side - half_thickness, center_y - half_
↪box_side};
        lines[1].pos = {center_x - half_box_side + half_thickness, center_y - half_
↪box_side};
        lines[2].pos = {center_x + half_box_side + half_thickness, center_y + half_
↪box_side};
        lines[3].pos = {center_x + half_box_side - half_thickness, center_y + half_
↪box_side};


        // Top-right to Bottom-left
        lines[4].pos = {center_x + half_box_side - half_thickness, center_y - half_
↪box_side};
        lines[5].pos = {center_x + half_box_side + half_thickness, center_y - half_
↪box_side};
        lines[6].pos = {center_x - half_box_side + half_thickness, center_y + half_
↪box_side};
        lines[7].pos = {center_x - half_box_side - half_thickness, center_y + half_
↪box_side};

        entity_registry.assign<geometry::vertex_array>(x_entity, lines,␣
↪geometry::vertex_geometry_type::quads);
        entity_registry.assign<entt::tag<"game_scene"_hs>>(x_entity);
        entity_registry.assign<entt::tag<"player_x"_hs>>(x_entity);
        entity_registry.assign<graphics::layer<1>>(x_entity);
    }

    void create_o(entt::registry &entity_registry, std::size_t row, std::size_t␣
↪column) noexcept
```

(continues on next page)

(continued from previous page)

```
    {
        auto constants = entity_registry.ctx<tic_tac_toe_constants>();
        const auto half_box_side = static_cast<float>(std::fmin(constants.cell_width,␣
↪constants.cell_height) * 0.25f);
        const auto center_x = static_cast<float>(constants.cell_width * 0.5 + column␣
↪* constants.cell_width);
        const auto center_y = static_cast<float>(constants.cell_height * 0.5 + row *␣
↪constants.cell_height);

        auto o_entity = geometry::blueprint_circle(entity_registry, half_box_side,␣
↪graphics::transparent,
                                                   transform::position_2d(center_x,␣
↪center_y),
                                                   graphics::outline_color(constants.
↪grid_thickness, graphics::cyan));

        entity_registry.assign<entt::tag<"game_scene"_hs>>(o_entity);
        entity_registry.assign<graphics::layer<1>>(o_entity);
    }
}

class tic_tac_toe_logic final : public ecs::logic_update_system<tic_tac_toe_logic>
{
public:

    void update() noexcept final
    {}

    void on_mouse_button_pressed(const event::mouse_button_pressed &evt) noexcept
    {
        if (current_game_state_ == running) {
            //! Retrieve game constants.
            auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

            //! Play one turn of the Tic-Tac-Toe
            play_turn(evt.y / constants.cell_height, evt.x / constants.cell_width);
        } else {
            //! Reset the game
            this->dispatcher_.trigger<reset_event>();
        }
    }

    tic_tac_toe_logic(entt::registry &registry, entt::entity grid_entity, std::vector
↪<cell_state> board) noexcept
            : system(registry), grid_entity_(grid_entity), board_(std::move(board))
    {
        //! Stateless system
        this->disable();

        //! Subscribe to mouse_button event
        this->dispatcher_.sink<event::mouse_button_pressed>().connect<&tic_tac_toe_
↪logic::on_mouse_button_pressed>(
                *this);
    }

    ~tic_tac_toe_logic() noexcept final
    {
```

(continues on next page)

(continued from previous page)

```
        this->dispatcher_.sink<event::mouse_button_pressed>().disconnect(*this);
    }

private:
    //! Game logic
    [[nodiscard]] bool did_current_player_win_the_game() const noexcept
    {
        std::size_t row_count{0u}, column_count{0u}, diag1_count{0u}, diag2_count{0u};
        auto[nb_cells, cell_width, cell_height, _] = entity_registry_.ctx<tic_tac_toe_
↪constants>();
        for (std::size_t i = 0; i < nb_cells; ++i) {
            for (std::size_t j = 0; j < nb_cells; ++j) {
                //! Check rows
                if (board_[i * nb_cells + j] == static_cast<cell_state>(player_turn_))
                    row_count++;

                //! Check columns
                if (board_[j * nb_cells + i] == static_cast<cell_state>(player_turn_))
                    column_count++;
            }

            //! Check condition
            if (row_count >= nb_cells || column_count >= nb_cells) {
                return true;
            }

            //! Reset rows and columns
            row_count = 0u, column_count = 0u;

            //! Diag1 count
            if (board_[i * nb_cells + i] == static_cast<cell_state>(player_turn_))
                diag1_count++;

            //! Second diag count
            if (board_[i * nb_cells + nb_cells - i - 1] == static_cast<cell_state>
↪(player_turn_))
                diag2_count++;
        }

        //! Condition
        return diag1_count >= nb_cells || diag2_count >= nb_cells;
    }

    [[nodiscard]] bool is_tie() const noexcept
    {
        return std::count(begin(board_), end(board_), cell_state::empty) == 0;
    }

    void check_winning_condition() noexcept
    {
        auto make_screen = [this](graphics::color clr_winner,
                                  entt::entity entity) {
            auto &array_cmp = this->entity_registry_.get<geometry::vertex_array>
↪(entity);
            for (auto &v : array_cmp.vertices) v.pixel_color = clr_winner;
            entity_registry_.replace<geometry::vertex_array>(entity, array_cmp.
↪vertices, array_cmp.geometry_type);
```

(continues on next page)

(continued from previous page)

```
        };

        auto make_player_win_screen = [this, make_screen](entt::entity entity) {
            auto winning_color = player_turn_ == player::x ? graphics::magenta :␣
↪graphics::cyan;
            make_screen(winning_color, entity);
        };

        auto make_tie_screen = [make_screen](entt::entity entity) {
            make_screen(graphics::yellow, entity);
        };

        if (did_current_player_win_the_game()) {
            current_game_state_ = static_cast<game_state>(player_turn_);
            make_player_win_screen(grid_entity_);
        } else if (is_tie()) {
            current_game_state_ = game_state::tie;
            make_tie_screen(grid_entity_);
        }
    }

    void play_turn(std::size_t row, std::size_t column) noexcept
    {
        //! Retrieve constants
        auto constants = entity_registry_.ctx<tic_tac_toe_constants>();

        //! Which cell is clicked ?
        std::size_t index = row * constants.nb_cells_per_axis + column;

        //! Cell is available ?
        if (index < board_.size() && board_[index] == cell_state::empty) {

            //! Change state of the cell to the current player
            board_[index] = static_cast<cell_state>(player_turn_);

            //! Create x or o based on the current player
            player_turn_ == x ? create_x(entity_registry_, row, column) : create_
↪o(entity_registry_, row, column);

            //! Check winning condition
            check_winning_condition();

            //! Switch player
            player_turn_ = (player_turn_ == player::x) ? player::o : player::x;
        }
    }

private:
    //! Private enums
    enum game_state
    {
        running,
        player_x_won = 1,
        player_y_won = 2,
        tie,
        quit
    };
```

(continues on next page)

(continued from previous page)

```
    enum player
    {
        x = 1,
        o = 2
    };

private:
    //! Private members variable
    entt::entity grid_entity_{entt::null};
    std::vector<cell_state> board_;
    game_state current_game_state_{game_state::running};
    player player_turn_{player::x};
};

//! Give a name to our system
REFL_AUTO(type(tic_tac_toe_logic));

class game_scene final : public scenes::base_scene
{
public:
    game_scene(entt::registry &entity_registry) noexcept : base_scene(entity_registry)
    {
        //! Subscribe to reset event
        this->dispatcher_.sink<reset_event>().connect<&game_scene::on_reset_event>
↪(*this);
        //! Retrieve canvas information
        auto[canvas_width, canvas_height] = entity_registry_.ctx<graphics::canvas_2d>
↪().canvas.size.to<math::vec2u>();

        //! Set the constants that will be used in the program
        auto &tictactoe_constants = entity_registry_.set<tic_tac_toe_constants>(3ull,␣
↪canvas_width, canvas_height);

        //! Create the grid of the tic tac toe
        auto grid_entity = create_grid(entity_registry_);

        //! Create the board of the tic tac toe
        auto board = create_board(tictactoe_constants.nb_cells_per_axis);

        //! Create our logic game system and give the fresh grid entity and the fresh␣
↪board.
        this->system_manager_.create_system<tic_tac_toe_logic>(grid_entity, board);
    }

    //! This function will not be used, because tic tac toe doesn't need an update␣
↪every frame.
    void update() noexcept final
    {}


    //! Our scene name
    std::string scene_name() noexcept final
    {
        return "game_scene";
    }
```

(continues on next page)

(continued from previous page)

```
    ~game_scene() noexcept final
    {
        //! Retrieve the collection of entities from the game scene
        auto view = entity_registry_.view<entt::tag<"game_scene"_hs>>();

        //! Iterate the collection and destroy each entities
        entity_registry_.destroy(view.begin(), view.end());

        //! Unset the tic tac toe constants
        entity_registry_.unset<tic_tac_toe_constants>();
    }

    //! Callback
    void on_reset_event(const reset_event &) noexcept
    {
        entt::registry &registry = this->entity_registry_;
        this->~game_scene();
        new(this) game_scene(registry);
    }

private:
    ecs::system_manager system_manager_{entity_registry_};
};

//! Our game world
struct tic_tac_toe_world : world::app
{
    //! Our game entry point
    tic_tac_toe_world() noexcept
    {
        //! Load our graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        //! Load our input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        //! Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        //! Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};

int main()
{
    //! Declare our world
    tic_tac_toe_world game;

    //! Run the game
    return game.run();
}
```

1.2.8 Tutorial: Flappy Bird

Before starting the tutorial, make sure that you have installed the required dependencies and your program compiles with the build commands available in the tutorial *getting started*.

This tutorial is divided into multiple steps to make it easier to follow.

Step 1: Setup the executable, window and the game scene

First, create a folder called `flappy-bird` for your project, and create a subfolder called `assets` inside. Within the `assets` folder create a `textures` subfolder (for storing the `player.png` image) Also within the `assets folder, create another subfolder called ``data`, and within this, create two subfolders: `linux` and `osx` to store utility files required to install the game on targeted systems.

In the `Linux` folder we need three files:

- komodo_icon.png (Icons of our game)
- org.antara.gaming.sfml.flappybird.appdata.xml (xml definition for our game)
- org.antara.gaming.sfml.flappybird.desktop (desktop file for Linux)

Here is the icon of the game that we will use for the tutorials:

Here is the xml file:

```
<component type="desktop-application">
    <id>org.antara.gaming.sfml.flappybird.desktop</id>
    <metadata_license>MIT</metadata_license>
    <project_license>MIT</project_license>
    <name>flappy-bird</name>
    <summary>flappy-bird tutorial antara gaming sdk</summary>
    <description>
        <p>Written in c++17</p>
    </description>
    <launchable type="desktop-id">org.antara.gaming.sfml.flappybird.desktop</
↪launchable>
    <url type="homepage">https://github.com/KomodoPlatform/antara-gaming-sdk</url>
    <screenshots>
        <screenshot type="default">
            <image>https://www.freedesktop.org/software/appstream/docs/images/scr-
↪examples/geany-good.png</image>
        </screenshot>
    </screenshots>
    <provides>
        <id>org.antara.gaming.sfml.flappybird.desktop</id>
    </provides>
</component>
```

Here is the desktop file:

```
[Desktop Entry]
Type=Application
Name=flappy-bird
Exec=flappy-bird
```

(continues on next page)

(continued from previous page)

```
Icon=komodo_icon
Categories=Game;
```

In the OSX folder we need four files:

- kmd_logo.icns (icon osx format of our game)
- Packaging_CMakeDMGBackground.tif (dmg image background)
- Packaging_CMakeDMGSetup.scpt (OSX Apple script for the packaging)
- sfml_flappybird_install.cmake (CMake script for the bundling)

Click here to download kmd_logo.icns.

Click here to download Packaging_CMakeDMGBackground.tif.

Here is the AppleScript:

```
on run argv
  set image_name to item 1 of argv

  tell application "Finder"
  tell disk image_name

    -- wait for the image to finish mounting
    set open_attempts to 0
    repeat while open_attempts < 4
      try
        open
          delay 1
          set open_attempts to 5
        close
      on error errStr number errorNumber
        set open_attempts to open_attempts + 1
        delay 10
      end try
    end repeat
    delay 5

    -- open the image the first time and save a DS_Store with just
    -- background and icon setup
    open
      set current view of container window to icon view
      set theViewOptions to the icon view options of container window
      set background picture of theViewOptions to file ".background:background.tif"
      set arrangement of theViewOptions to not arranged
      set icon size of theViewOptions to 128
      delay 5
    close

    -- next setup the position of the app and Applications symlink
    -- plus hide all the window decorationPackaging_CMakeDMGBackground.tif
    open
      update without registering applications
      tell container window
        set sidebar width to 0
        set statusbar visible to false
        set toolbar visible to false
```

(continues on next page)

(continued from previous page)

```
        set the bounds to { 400, 100, 900, 465 }
        set position of item "flappy-bird.app" to { 133, 200 }
        set position of item "Applications" to { 378, 200 }
      end tell
      update without registering applications
      delay 5
    close

    -- one last open and close so you can see everything looks correct
    open
      delay 5
    close

  end tell
  delay 1
end tell
end run
```

And the CMake script:

```
if (APPLE)
    set_target_properties(${PROJECT_NAME} PROPERTIES
            MACOSX_BUNDLE_BUNDLE_NAME "${PROJECT_NAME}"
            RESOURCE data/osx/${PROJECT_NAME}.icns
            MACOSX_BUNDLE_ICON_FILE ${PROJECT_NAME}
            MACOSX_BUNDLE_SHORT_VERSION_STRING 0.0.1
            MACOSX_BUNDLE_LONG_VERSION_STRING 0.0.1
            MACOSX_BUNDLE_INFO_PLIST "${PROJECT_SOURCE_DIR}/cmake/MacOSXBundleInfo.
↪plist.in")
    add_custom_command(TARGET ${PROJECT_NAME}
            POST_BUILD COMMAND
            ${CMAKE_INSTALL_NAME_TOOL} -add_rpath "@executable_path/../Frameworks/"
            $<TARGET_FILE:${PROJECT_NAME}>)
endif ()

if (APPLE)
    install(TARGETS ${PROJECT_NAME}
            BUNDLE DESTINATION . COMPONENT Runtime
            RUNTIME DESTINATION bin COMPONENT Runtime
            )

    # Note Mac specific extension .app
    set(APPS "\${CMAKE_INSTALL_PREFIX}/${PROJECT_NAME}.app")

    # Directories to look for dependencies
    set(DIRS ${CMAKE_BINARY_DIR})

    install(CODE "include(BundleUtilities)
    fixup_bundle(\"${APPS}\" \"\" \"${DIRS}\")")

    set(CPACK_GENERATOR "DRAGNDROP")
    set(CPACK_DMG_DS_STORE_SETUP_SCRIPT "${CMAKE_CURRENT_SOURCE_DIR}/data/osx/
↪Packaging_CMakeDMGSetup.scpt")
    set(CPACK_DMG_BACKGROUND_IMAGE "${CMAKE_CURRENT_SOURCE_DIR}/data/osx/Packaging_
↪CMakeDMGBackground.tif")
    set(CPACK_PACKAGE_NAME "${PROJECT_NAME}")
    include(CPack)
```

(continues on next page)

(continued from previous page)

```
endif ()
```

Now create a text file and save it as `CMakeLists.txt`.

In this `CMakeLists.txt` file we will have: name of the project, creation of the executable, link with the SDK, moving of the assets, C++ standard that will be used, and any extra modules that we need.

Below is the `CMakeLists.txt` file:

```
if (${CMAKE_SOURCE_DIR} STREQUAL ${CMAKE_BINARY_DIR})
    message(FATAL_ERROR "Prevented in-tree build. Please create a build directory␣
↪outside of the source code and call cmake from there")
endif ()

##! Minimum version of the CMake.
cmake_minimum_required(VERSION 3.14)

##! C++ Standard needed by the SDK is 17
set(CMAKE_CXX_STANDARD 17)

##! Our Project title, here flappy-bird.
project(flappy-bird DESCRIPTION "An awesome flappy-bird" LANGUAGES CXX VERSION 1.0.0)

##! The SDK need's clang as main compiler.
if (NOT "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
    if (NOT "${CMAKE_CXX_COMPILER_ID}" STREQUAL "AppleClang")
        message(FATAL_ERROR "Only Clang is supported (minimum LLVM 8.0)")
    endif()
endif ()

##! We will let know the SDK if our on Linux
if (${CMAKE_SYSTEM_NAME} STREQUAL "Linux")
    set(LINUX TRUE)
endif ()

##! We include the module from CMake for fetching dependencies
include(FetchContent)

##! We declare information about the dependance that we want to fetch.
FetchContent_Declare(
        antara-gaming-sdk
        URL https://github.com/KomodoPlatform/antara-gaming-sdk/archive/master.zip
)

##! We set extras modules from the SDK that we want to use, here we will use the SFML␣
↪module.
set(USE_SFML_ANTARA_WRAPPER ON)

##! We fetch our dependence
FetchContent_MakeAvailable(antara-gaming-sdk)

##! Calling this macros provided by the sdk will if you are on Apple init the␣
↪environment for this OS (std::filesystem).
init_antara_env()

##! Get basis assets (default fonts, etc)
get_resources_basics_assets(${CMAKE_CURRENT_SOURCE_DIR})
```

(continues on next page)

(continued from previous page)

```
##! Osx bundle icon
set(ICON)
configure_icon_osx(data/osx/kmd_logo.icns ICON kmd_logo.icns)

##! We create the executable with the project name
add_executable(${PROJECT_NAME} MACOSX_BUNDLE ${ICON} flappy-bird.cpp)

##! Linux assets
magic_game_app_image_generation("${CMAKE_CURRENT_SOURCE_DIR}/data/linux"
        "org.antara.gaming.sfml.flappybird.desktop"
        "org.antara.gaming.sfml.flappybirds.appdata.xml"
        "komodo_icon.png"
        flappy-bird
        AntaraFlappyBirdAppDir
        ${CMAKE_CURRENT_SOURCE_DIR}/assets
        )

##! Setting output directory
set_target_properties(${PROJECT_NAME}
        PROPERTIES
        RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin/"
        )

##! We link the SDK modules that we want to use to our executable
target_link_libraries(${PROJECT_NAME} PUBLIC antara::world antara::sfml␣
↪antara::collisions)

##! Move assets
if (WIN32)
    file(COPY assets DESTINATION ${CMAKE_BINARY_DIR}/bin/)
    ADD_CUSTOM_COMMAND(TARGET ${PROJECT_NAME} POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy_directory "${SFML_BINARY_DIR}/lib" "$
↪{CMAKE_BINARY_DIR}/bin/"
            COMMENT "copying dlls ..."
            $<TARGET_FILE_DIR:${PROJECT_NAME}>
            )

    ADD_CUSTOM_COMMAND(TARGET ${PROJECT_NAME} POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy "${SFML_SOURCE_DIR}/extlibs/bin/x64/
↪openal32.dll" "${CMAKE_BINARY_DIR}/bin/openal32.dll"
            COMMENT "copying dlls ..."
            $<TARGET_FILE_DIR:${PROJECT_NAME}>
            )
endif ()

if (APPLE)
    file(COPY assets DESTINATION ${CMAKE_BINARY_DIR}/bin/${PROJECT_NAME}.app/Contents/
↪Resources)
endif()
```

Now we can create our input file for the application containing an empty main function (as below) and save it as `flappy-bird.cpp`.:

```
int main() {
    return 0;
}
```

You should now have the following tree:

```
./flappy-bird
├── assets
│   ├── config
│   │   └── game.config.maker.json
│   └── textures
│       └── player.png
│   └── fonts
│       └── sansation.ttf
├── CMakeLists.txt
├── data
│   ├── linux
│   │   ├── komodo_icon.png
│   │   ├── org.antara.gaming.sfml.flappybird.appdata.xml
│   │   └── org.antara.gaming.sfml.flappybird.desktop
│   └── osx
│       ├── kmd_logo.icns
│       ├── Packaging_CMakeDMGBackground.tif
│       ├── Packaging_CMakeDMGSetup.scpt
│       └── sfml_flappybird_install.cmake
└── flappy-bird.cpp
```

Now we need a world representing the world of our game, to do this we use the following header file: `#include <antara/gaming/world/world.app.hpp>`

And a basic structure that we name `flappy_bird_world`. It will inherit from `antara::gaming::world::app` class.

And use the namespace `antara::gaming` and `std::string_literals` to make things easier.

Finally, we declare our new object in the body of the main function, and we replace its return value with the return value of our game (returned by the `run` function of the `class world::app`).

It gives us the following result:

```cpp
#include <antara/gaming/world/world.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept = default;
};

int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

If you compile now and run your executable, you have an infinite loop and nothing will happen.

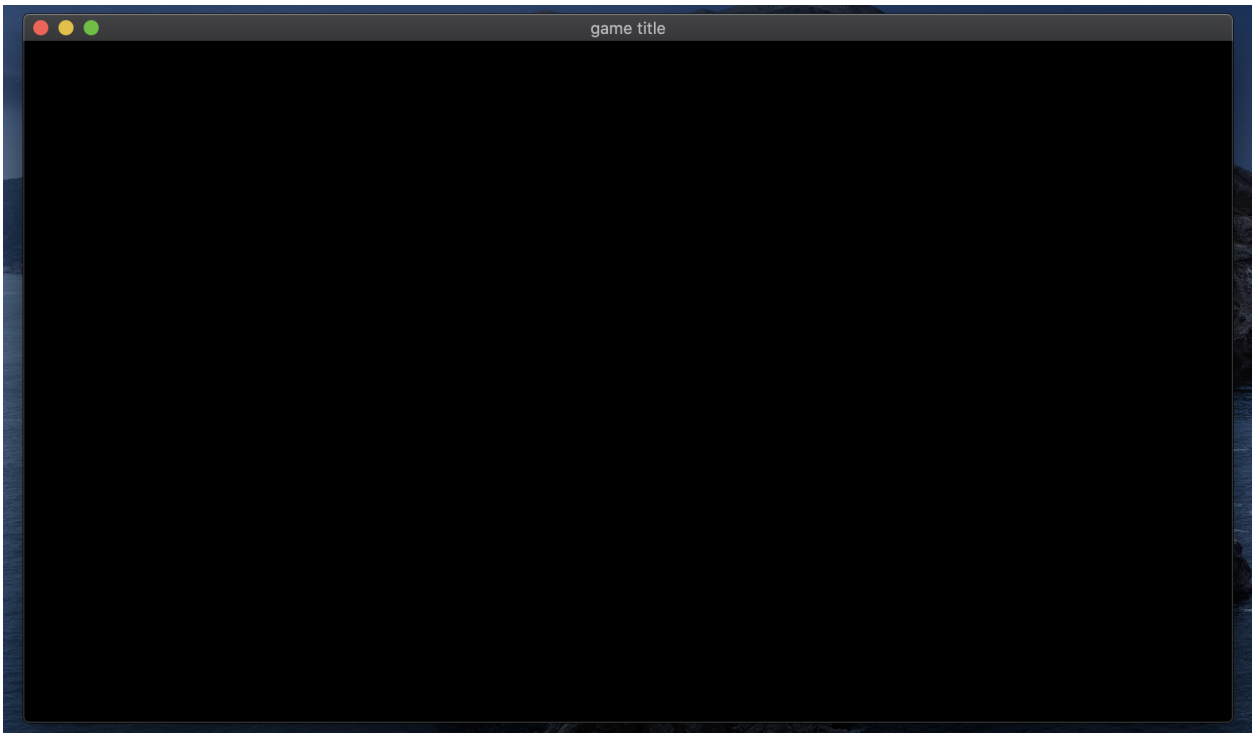
The last stage of this step one is to add the graphics side of the application, for that we will need two modules: `antara::gaming::sfml::graphic_system` and `antara::gaming::sfml::input::system`

which have these following headers, respectively: `#include <antara/gaming/sfml/graphic.system.hpp>` and `#include <antara/gaming/sfml/input.system.hpp>`.

Now in the body of the constructor of our class `flappy_bird_world` we will load the graphic system, then initialize the input system with the window from the graphic system.

```
// Game entry point
flappy_bird_world() noexcept {
    // Load the graphical system
    auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

    // Load the input system with the window from the graphical system
    system_manager_.create_system<sfml::input_system>(graphic_system.get_window());
}
```

If you compile and run now, you should see a black window open. You can close by pressing the close button of the window:



And now, the setup part is over. We have a `CMakeLists.txt` to be able to compile our program into a basic executable which can create the game window.

Next we create a game scene using the scene manager. To do so we need to include the header file `#include <antara/gaming/scenes/scene.manager.hpp>` and load the scenes manager system into the system manager.

```
// Game entry point
struct flappy_bird_world : world::app {
    //! Our game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the input system with the window from the graphical system
```

(continues on next page)

(continued from previous page)

```
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();
    }
};
```

Now we create the `game_scene` class that inherits from the `base_scene` class. This class will be the entry point of our game scene.

The concrete class must override several functions such as update, `scene_name`. Flappy Bird is a game that needs an update for each tick, so we will fill the update function later. For the `scene_name` function we'll just return the name of the scene.

```
// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry) noexcept : base_scene(registry) {
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
    }
};
```

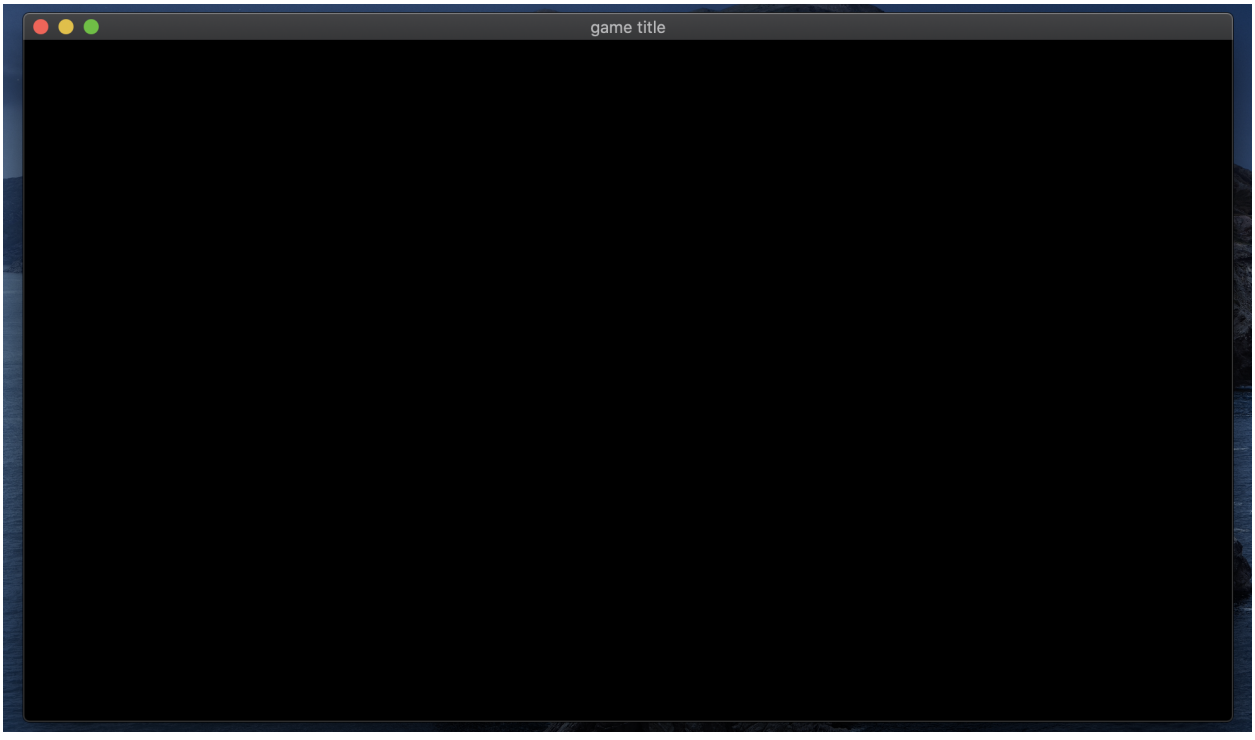
Now we load our game scene into the `scene_manager` using the `change_scene` member function.

```
struct flappy_bird_world : world::app
{
    //! Our game entry point
    flappy_bird_world() noexcept
    {
        //! Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        //! Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        //! Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};
```

We will also use a sprite for the bird, so we need the `sfml::resources_system` by including the header file `#include <antara/gaming/sfml/resources.manager.hpp>` and loading it in the world constructor.

```
// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};
```

If you compile now you should still see the black window from the previous step, but we are now in our game scene.



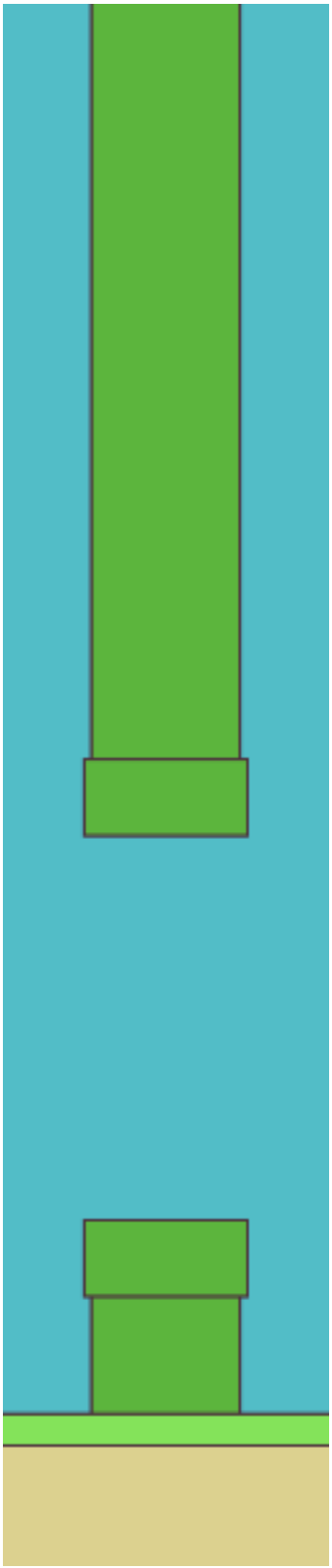
Note: The scene system is very handy to organize multiple screens of the game: **introduction scene**, **game scene**, **end-of-game scene**, etc.

Step 1 is complete, here is the full code.

```
#include <random>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry) noexcept : base_scene(registry) {
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
    }
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};

int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

Step 2: Creation of Pipes

During this step, we will add the pipes which kill Flappy Bird when it touches them. In the image below, you'll see two pipes with a gap between them. We will call this a `column`.



Let's start with the constant values that we will use. We will keep them in a struct. There are many of them:

```
// Constants
struct flappy_bird_constants {
    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};
};
```

Then we will add this to the `registry` in the `game_scene` constructor.

```
// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry) noexcept : base_scene(registry) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();
    }
```

Now we'll make a struct which will represent a single `pipe`. Instead of using a sprite, we will make graphics with basic shapes. For example, a pipe has two parts as you see in the image above: `body` and `cap`. The body is the long part of the pipe with a cap sitting at the tip. Both will be green rectangle entities but with different sizes. We also prepare a `destroy` function which will destroy `body` and `cap` entities.

```
// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
    entt::entity cap{entt::null};

    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};
```

As mentioned before, two of these pipes will be called a `column`. Here we make another struct which uses the `struct pipe`. One is `top_pipe`, another one is `bottom_pipe`. Again, we have the `destroy` function, but this time the `destroy` function also has an entity parameter which will be the `column` entity itself.

```
// Column is made of two pipes
struct column {
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
        top_pipe.destroy(registry);
```

(continues on next page)

(continued from previous page)

```
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
};
```

We will need some functions for creation of the pipes. First one is a function which returns a random number, we will use this to randomly position the gap between the pipes. We will use `std::random_device`, `std::mt19937` and `std::uniform_real_distribution<float>` for this.

```
// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}
```

We will have many entities, so we need to tag them with `game_scene` name. Dynamic entities will have a `dynamic` tag, so we can easily query the dynamic ones to destroy them at game reset. Since this tagging will be repeated a lot, we will create a function for it. It is also a good idea to have these kind of helper functions in a namespace.

```
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }
}
```

During the creation of the pipes we will need another function, to get a random starting position of the gap. That's how we will know where to start and end the top pipe, have a gap, then start and end the bottom pipe.

This function will also use some constants, such as `column_min` and `column_max`. `column_min` is for the top limit, `0.2` of the canvas height. And `column_max` is for the bottom limit, `0.8` of the canvas height. Though we also need to subtract `gap_height` from the `bottom_limit` because this will be the starting position (or top position) of the gap. When the limits are set, function returns a random float value between those two, using the random function we defined earlier. We will add this function into the same namespace.

```
// Returns a random gap start position Y
float get_random_gap_start_pos(const entt::registry &registry) {
    // Retrieve constants
    const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.y();
    const auto constants = registry.ctx<flappy_bird_constants>();

    float top_limit = canvas_height * constants.column_min;
    float bottom_limit = canvas_height * constants.column_max - constants.gap_height;

    return random_float(top_limit, bottom_limit);
}
```

Now we can start constructing a pipe. There will be some math here about position and size.

`create_pipe` function will have `bool is_top, float pos_x, float gap_start_pos_y` parameters. `is_top` indicates if it's the top pipe or the bottom. `pos_x` is the horizontal position of the pipe. `gap_start_pos_y` is the vertical start position of the gap, for example, the bottom edge of the top pipe.

To start with, we retrieve `canvas_height` and the constants.

```
// Retrieve constants
const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.y();
const auto constants = registry.ctx<flappy_bird_constants>();
```

Remember that pipe is made of two parts: body and cap. Let's construct the body first. It will be a rectangle so we will need center position and size. Just to avoid more complicated math, we can have center of the rectangle at the screen edge. Half of the pipe will be out of the view but it for a basic example we can ignore that visual optimization.

X will be `pos_x`, and the Y will be top of the screen if it's the top pipe, which is 0. If it's a bottom one, then Y will be bottom edge of the screen, which is `canvas_height`.

```
// PIPE BODY
// Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the canvas
transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};
```

Body size however, is a little tricky. Size X will be the column thickness, that's easy. But the Size Y changes depending on if it's the top pipe or the bottom.

If it's the top pipe, start of the gap `gap_start_pos_y` should be bottom of the rectangle. So half size should be `gap_start_pos_y` since the center of the rectangle is at 0. Full size will be `gap_start_pos_y * 2.0f`.

If it's the bottom pipe, top of the rectangle will be the end of the gap, `gap_start_pos_y + gap_height`. So half size should be `canvas_height - (gap_start_pos_y + gap_height)`. And we need to double it for the full size. That makes `(canvas_height - (gap_start_pos_y + constants.gap_height)) * 2.0f`.

```
// Size X is the column thickness,
// Size Y is the important part.
// If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
//  So half size should be gap_start_pos_y since center of the rectangle is at 0.
// If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y + gap_
↪height
//  So half size should be canvas_height - (gap_start_pos_y + gap_height)
// Since these are half-sizes, and the position is at the screen border, we multiply␣
↪these sizes by two
math::vec2f body_size{constants.column_thickness,
                      is_top ?
                      gap_start_pos_y * 2.0f :
                      (canvas_height - (gap_start_pos_y + constants.gap_height)) *␣
↪2.0f};
```

To construct the rectangle entity, we can use blueprint function `geometry::blueprint_rectangle`. We will also feed `pipe_color` and `pipe_outline_color` (which includes line thickness information).

```
auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_color,␣
↪body_pos, constants.pipe_outline_color);
```

That's it for the body! Now we need to construct the cap of the pipe.

Size of the cap will be `column_thickness` plus `pipe_cap_extra_width` because we want the cap to look like the mario pipe, so the cap needs to be a little bit wider. The height is pre-defined `pipe_cap_height`. Easy!

```
// PIPE CAP
// Let's prepare the pipe cap
// Size of the cap is defined in constants
math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_width,␣
↪constants.pipe_cap_height};
```

Cap position is a bit trickier, X position will be same as the body, `body_pos.x()` since it's centered. But the Y position changes depending on if it's a top one or bottom.

If it's the top cap, bottom line of the cap is alligned with bottom of the body or start of the gap which is the same line. We will use start of the gap here, minus half of the cap height, because position is the center of the rectangle. It makes `gap_start_pos_y - constants.pipe_cap_height * 0.5f`.

If it's the bottom cap, bottom of the gap will be the same line as top of the cap. Bottom of the gap is gap start position plus the gap height, `gap_start_pos_y + constants.gap_height`. Then we need to add half of the pipe height again because we want the shift a little bit down since the position we define is the center of the cap and we want the top to be alligned with top of the body.

```
// Position, X is same as the body. Bottom of the cap is aligned with bottom of the␣
↪body,
// or start of the gap, we will use start of the gap here, minus half of the cap␣
↪height
transform::position_2d cap_pos{body_pos.x(),
                               is_top ?
                               gap_start_pos_y - constants.pipe_cap_height * 0.5f :
                               gap_start_pos_y + constants.gap_height + constants.
↪pipe_cap_height * 0.5f
};
```

To construct the rectangle entity, we can use blueprint function `geometry::blueprint_rectangle`. We will also feed `pipe_color` and `pipe_outline_color` again, with color set to the same as the body.

```
auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_color,␣
↪cap_pos, constants.pipe_outline_color);
```

To make the cap appear in front of the body, we need to define the draw order. We will use `graphics::layer` for that. Higher is front, lower is back. We set `cap` as `layer<4>` and `body` as `layer<3>`.

```
// Set layers, cap should be in front of body
registry.assign<graphics::layer<4>>(cap);
registry.assign<graphics::layer<3>>(body);
```

Now tag both entities as `game_scene` and `dynamic` with the `tag_game_scene` function we defined earlier, then return both inside `{ }` to automatically construct a `struct pipe`.

```
tag_game_scene(registry, cap, true);
tag_game_scene(registry, body, true);

// Construct a pipe with body and cap and return it
return {body, cap};
```

The completed function looks like this:

```
// Factory for pipes, requires to know if it's a top one, position x of the column,␣
↪and the gap starting position Y
pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_start_
↪pos_y) {
```

(continues on next page)

(continued from previous page)

```
    // Retrieve constants
    const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.y();
    const auto constants = registry.ctx<flappy_bird_constants>();

    // PIPE BODY
    // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the canvas
    transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

    // Size X is the column thickness,
    // Size Y is the important part.
    // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
    //  So half size should be gap_start_pos_y since center of the rectangle is at 0.
    // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y + gap_
↪height
    //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
    // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
    math::vec2f body_size{constants.column_thickness,
                          is_top ?
                          gap_start_pos_y * 2.0f :
                          (canvas_height - (gap_start_pos_y + constants.gap_height))␣
↪* 2.0f};

    auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                              constants.pipe_outline_color);

    // PIPE CAP
    // Let's prepare the pipe cap
    // Size of the cap is defined in constants
    math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_width,␣
↪constants.pipe_cap_height};

    // Position, X is same as the body. Bottom of the cap is aligned with bottom of␣
↪the body,
    // or start of the gap, we will use start of the gap here, minus half of the cap␣
↪height
    transform::position_2d cap_pos{body_pos.x(),
                                   is_top ?
                                   gap_start_pos_y - constants.pipe_cap_height * 0.5f␣
↪:
                                   gap_start_pos_y + constants.gap_height + constants.
↪pipe_cap_height * 0.5f
    };

    // Construct the cap
    auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_color,
↪ cap_pos,
                                             constants.pipe_outline_color);

    // Set layers, cap should be in front of body
    registry.assign<graphics::layer<4>>(cap);
    registry.assign<graphics::layer<3>>(body);
    tag_game_scene(registry, cap, true);
    tag_game_scene(registry, body, true);

    // Construct a pipe with body and cap and return it
```

(continues on next page)

(continued from previous page)

```
    return {body, cap};
}
```

Since we are able to make a single pipe now, we can use it to build a full column which (two pipes and a gap).

Let's make the function `void create_column(entt::registry &registry, float pos_x)`, with a single parameter `pos_x` for the X position of the column.

We start with creating an empty entity.

```
// Create a fresh entity for a new column
auto entity_column = registry.create();
```

Then we get a random vertical position for start of the gap with the function we created earlier, `get_random_gap_start_pos`.

```
// Get a random gap start position Y, between pipes
float gap_start_pos_y = get_random_gap_start_pos(registry);
```

Create `top_pipe` and `bottom_pipe` with the `create_pipe` function. The only parameter that varies between the two being `is_top` boolean is true for the `top_pipe`, and false for `bottom_pipe`.

```
// Create pipes, is_top variable is false for bottom one
auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);
```

Now we can construct a `struct column` with these two, tag it with `column` name, then use the `tag_game_scene` function to tag it with `game_scene` and `dynamic`.

```
// Make a column from these two pipes and mark it as "column"
registry.assign<column>(entity_column, top_pipe, bottom_pipe);
registry.assign<entt::tag<"column"_hs>>(entity_column);
tag_game_scene(registry, entity_column, true);
```

The completed function looks like this:

```
// Factory to create single column
void create_column(entt::registry &registry, float pos_x) noexcept {
    // Create a fresh entity for a new column
    auto entity_column = registry.create();

    // Get a random gap start position Y, between pipes
    float gap_start_pos_y = get_random_gap_start_pos(registry);

    // Create pipes, is_top variable is false for bottom one
    auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
    auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

    // Make a column from these two pipes and mark it as "column"
    registry.assign<column>(entity_column, top_pipe, bottom_pipe);
    registry.assign<entt::tag<"column"_hs>>(entity_column);
    tag_game_scene(registry, entity_column, true);
}
```

We want many of these columns, which we can do with a `create_columns` function.

To start, we will need constants again, so we retrieve them.

```
// Retrieve constants
const auto constants = registry.ctx<flappy_bird_constants>();
```

Columns move towards Flappy Bird start from a distance, so we have `column_start_distance` to add that offset, and an additional `constants.column_thickness * 2.0f` to make sure they are out of the screen if `column_start_distance` is set as `canvas_width`.

```
// Spawn columns far away
const float column_pos_offset = constants.column_start_distance + constants.column_
↪thickness * 2.0f;
```

Using the `create_column` function in a `for loop` we can easily create more columns. For count, we use `column_count` constant, and to add distance between every column, we can use the counter `i` which increments by one, multiplying `i` with `column_distance` puts each column further than the previous one. Finally, add the `column_pos_offset` offset and the loop will look like this:

```
// Create the columns
for (std::size_t i = 0; i < constants.column_count; ++i) {
    // Horizontal position (X) increases for every column, keeping the distance
    float pos_x = column_pos_offset + i * constants.column_distance;

    create_column(registry, pos_x);
}
```

The completed function looks like this:

```
// Factory for creating a Flappy Bird columns
void create_columns(entt::registry &registry) noexcept {
    // Retrieve constants
    const auto constants = registry.ctx<flappy_bird_constants>();

    // Spawn columns out of the screen, out of the canvas
    const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

    // Create the columns
    for (std::size_t i = 0; i < constants.column_count; ++i) {
        // Horizontal position (X) increases for every column, keeping the distance
        float pos_x = column_pos_offset + i * constants.column_distance;

        create_column(registry, pos_x);
    }
}
```

We will call this `create_columns` function at initialization. Let's make an initialization function for dynamic objects.

```
// Initialize dynamic objects, this function is called at start and resets
void init_dynamic_objects(entt::registry &registry) {
    create_columns(registry);
}
```

And call it in the `game_scene` constructor:

```
game_scene(entt::registry &registry) noexcept : base_scene(registry) {
    // Set the constants that will be used in the program
    registry.set<flappy_bird_constants>();
```

(continues on next page)

(continued from previous page)

```
    // Create everything
    init_dynamic_objects(registry);
}
```

That's it! Now we have many columns being drawn:

Step 2 is complete, here is the full code.

```
#include <random>
#include <antara/gaming/graphics/component.layer.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/graphics/component.sprite.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Constants
struct flappy_bird_constants {
    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};
};

// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}

// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
    entt::entity cap{entt::null};

    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};

// Column is made of two pipes
struct column {
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
```

(continues on next page)

(continued from previous page)

```
        top_pipe.destroy(registry);
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
};

// Logic functions
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }

    // Returns a random gap start position Y
    float get_random_gap_start_pos(const entt::registry &registry) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        float top_limit = canvas_height * constants.column_min;
        float bottom_limit = canvas_height * constants.column_max - constants.gap_
↪height;

        return random_float(top_limit, bottom_limit);
    }
}

// Factory functions
namespace {
    // Factory for pipes, requires to know if it's a top one, position x of the␣
↪column, and the gap starting position Y
    pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_
↪start_pos_y) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        // PIPE BODY
        // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the␣
↪canvas
        transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

        // Size X is the column thickness,
        // Size Y is the important part.
        // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
        //  So half size should be gap_start_pos_y since center of the rectangle is␣
↪at 0.
        // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y +␣
↪gap_height
        //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
        // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
```

(continues on next page)

(continued from previous page)

```
        math::vec2f body_size{constants.column_thickness,
                              is_top ?
                              gap_start_pos_y * 2.0f :
                              (canvas_height - (gap_start_pos_y + constants.gap_
↪height)) * 2.0f};

        auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                                  constants.pipe_outline_color);

        // PIPE CAP
        // Let's prepare the pipe cap
        // Size of the cap is defined in constants
        math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_
↪width, constants.pipe_cap_height};

        // Position, X is same as the body. Bottom of the cap is aligned with bottom␣
↪of the body,
        // or start of the gap, we will use start of the gap here, minus half of the␣
↪cap height
        transform::position_2d cap_pos{body_pos.x(),
                                       is_top ?
                                       gap_start_pos_y - constants.pipe_cap_height *␣
↪0.5f :
                                       gap_start_pos_y + constants.gap_height +␣
↪constants.pipe_cap_height * 0.5f
        };

        // Construct the cap
        auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_
↪color, cap_pos,
                                                 constants.pipe_outline_color);

        // Set layers, cap should be in front of body
        registry.assign<graphics::layer<4>>(cap);
        registry.assign<graphics::layer<3>>(body);
        tag_game_scene(registry, cap, true);
        tag_game_scene(registry, body, true);

        // Construct a pipe with body and cap and return it
        return {body, cap};
    }

    // Factory to create single column
    void create_column(entt::registry &registry, float pos_x) noexcept {
        // Create a fresh entity for a new column
        auto entity_column = registry.create();

        // Get a random gap start position Y, between pipes
        float gap_start_pos_y = get_random_gap_start_pos(registry);

        // Create pipes, is_top variable is false for bottom one
        auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
        auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

        // Make a column from these two pipes and mark it as "column"
        registry.assign<column>(entity_column, top_pipe, bottom_pipe);
```

(continues on next page)

(continued from previous page)

```
        registry.assign<entt::tag<"column"_hs>>(entity_column);
        tag_game_scene(registry, entity_column, true);
    }

    // Factory for creating a Flappy Bird columns
    void create_columns(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Spawn columns out of the screen, out of the canvas
        const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

        // Create the columns
        for (std::size_t i = 0; i < constants.column_count; ++i) {
            // Horizontal position (X) increases for every column, keeping the␣
↪distance
            float pos_x = column_pos_offset + i * constants.column_distance;

            create_column(registry, pos_x);
        }
    }
}

// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry) noexcept : base_scene(registry) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();

        // Create everything
        init_dynamic_objects(registry);
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
    }

    // Initialize dynamic objects, this function is called at start and resets
    void init_dynamic_objects(entt::registry &registry) {
        create_columns(registry);
    }
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();
```

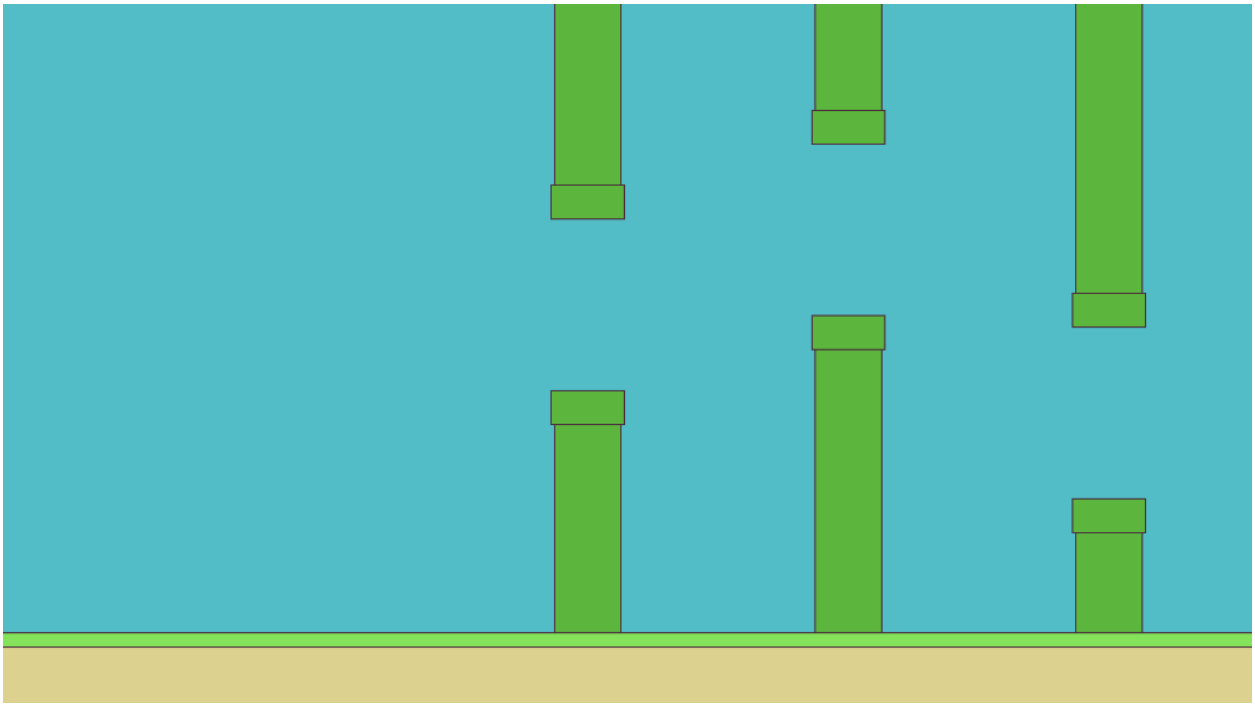
(continues on next page)

(continued from previous page)

```
        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};

int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

Step 3: Creation of the background

Now it is time to maker the black background prettier by adding sky, ground and grass. This is how we want it to look like:



Let's add the constants to `struct flappy_bird_constants`: thickness and colors.

```
// Background
const float ground_thickness{100.0f};
const float grass_thickness{20.0f};
const graphics::color background_color{82, 189, 199};
const graphics::color ground_color{220, 209, 143};
const graphics::color grass_color{132, 227, 90};
const graphics::outline_color grass_outline_color{2.0f, graphics::color{76, 47, 61}};
```

Now we make a function called `create_background`.

First retrieve the constants and canvas size:

```
// Retrieve constants
const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.
↪size;
const auto constants = registry.ctx<flappy_bird_constants>();
```

Let's create the sky next, a simple blue rectangle.

Position is center of the canvas.

```
// Sky is whole canvas so position is middle of it
transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};
```

Set the size to whole canvas (other visual elements will appear in the foreground).

```
// And the size is full canvas
math::vec2f size{canvas_width, canvas_height};
```

Use the `geometry::blueprint_rectangle` again and use `background_color` from defined constants.

```
auto sky = geometry::blueprint_rectangle(registry, size, constants.background_color,␣
↪pos);
```

Set it to appear at `layer<1>` and tag `game_scene`:

```
registry.assign<graphics::layer<1>>(sky);
tag_game_scene(registry, sky);
```

Here is how the whole sky creation snippet looks like:

```
// Create Sky
{
    // Sky is whole canvas so position is middle of it
    transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

    // And the size is full canvas
    math::vec2f size{canvas_width, canvas_height};

    auto sky = geometry::blueprint_rectangle(registry, size, constants.background_
↪color, pos);
    registry.assign<graphics::layer<1>>(sky);
    tag_game_scene(registry, sky);
}
```

Now we do the same thing, but for grass. X position is middle of the canvas, Y position is canvas height minus ground thickness because grass grows above the ground!

```
// Ground expands to whole canvas width so position is middle of it,
// But position Y is at top of the ground, so it's canvas height minus ground␣
↪thickness
transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.ground_
↪thickness};
```

Size Y is constant `grass_thickness` and Size X is full `canvas_width` plus the outline thickness because we don't wanna see the left and right edges, by making it bigger edges will be out of the canvas.

```
// Size X is full canvas but the height is defined in constants
// We also make it a bit longer by adding the thickness of the outline to hide the␣
↪outline at sides
math::vec2f size{canvas_width + constants.grass_outline_color.thickness * 2.0f,␣
↪constants.grass_thickness};
```

We use `geometry::blueprint_rectangle` again, and assign `layer<3>`, then tag it.

```
auto grass = geometry::blueprint_rectangle(registry, size, constants.grass_color, pos,
↪ constants.grass_outline_color);
registry.assign<graphics::layer<3>>(grass);
tag_game_scene(registry, grass);
```

Here is the completed grass creation snippet looks:

```
// Create Grass
{
    // Ground expands to whole canvas width so position is middle of it,
    // But position Y is at top of the ground, so it's canvas height minus ground␣
↪thickness
    transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.ground_
↪thickness};

    // Size X is full canvas but the height is defined in constants
    // We also make it a bit longer by adding the thickness of the outline to hide␣
↪the outline at sides
    math::vec2f size{canvas_width + constants.grass_outline_color.thickness * 2.0f,␣
↪constants.grass_thickness};

    auto grass = geometry::blueprint_rectangle(registry, size, constants.grass_color,␣
↪pos, constants.grass_outline_color);
    registry.assign<graphics::layer<3>>(grass);
    tag_game_scene(registry, grass);
}
```

Now we create the ground, it's easy.

X position is middle of canvas, height is canvas height minus half of the ground thickness (because position is center of the rectangle).

Size X is canvas width, Size Y is ground thickness.

```
// Create Ground
{
    // Ground expands to whole canvas width so position is middle of it,
    // But position Y is at bottom of the screen so it's full canvas_height minus␣
↪half of the ground thickness
    transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.ground_
↪thickness * 0.5f};
```

(continues on next page)

(continued from previous page)

```
    // Size X is full canvas but the height is defined in constants
    math::vec2f size{canvas_width, constants.ground_thickness};

    auto ground = geometry::blueprint_rectangle(registry, size, constants.ground_
↪color, pos);
    registry.assign<graphics::layer<3>>(ground);
    tag_game_scene(registry, ground);
}
```

Notice that we didn't tag any of these `dynamic`. They will be static and permanent.

Now the background is complete, the whole function looks like this:

```
// Factory for creating a Flappy Bird background
void create_background(entt::registry &registry) noexcept {
    // Retrieve constants
    const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
    const auto constants = registry.ctx<flappy_bird_constants>();

    // Create Sky
    {
        // Sky is whole canvas so position is middle of it
        transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

        // And the size is full canvas
        math::vec2f size{canvas_width, canvas_height};

        auto sky = geometry::blueprint_rectangle(registry, size, constants.background_
↪color, pos);
        registry.assign<graphics::layer<1>>(sky);
        tag_game_scene(registry, sky);
    }

    // Create Grass
    {
        // Ground expands to whole canvas width so position is middle of it,
        // But position Y is at top of the ground, so it's canvas height minus ground␣
↪thickness
        transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness};

        // Size X is full canvas but the height is defined in constants
        // We also make it a bit longer by adding the thickness of the outline to␣
↪hide the outline at sides
        math::vec2f size{canvas_width + constants.grass_outline_color.thickness * 2.
↪0f, constants.grass_thickness};

        auto grass = geometry::blueprint_rectangle(registry, size, constants.grass_
↪color, pos,
                                                   constants.grass_outline_color);
        registry.assign<graphics::layer<3>>(grass);
        tag_game_scene(registry, grass);
    }

    // Create Ground
```

(continues on next page)

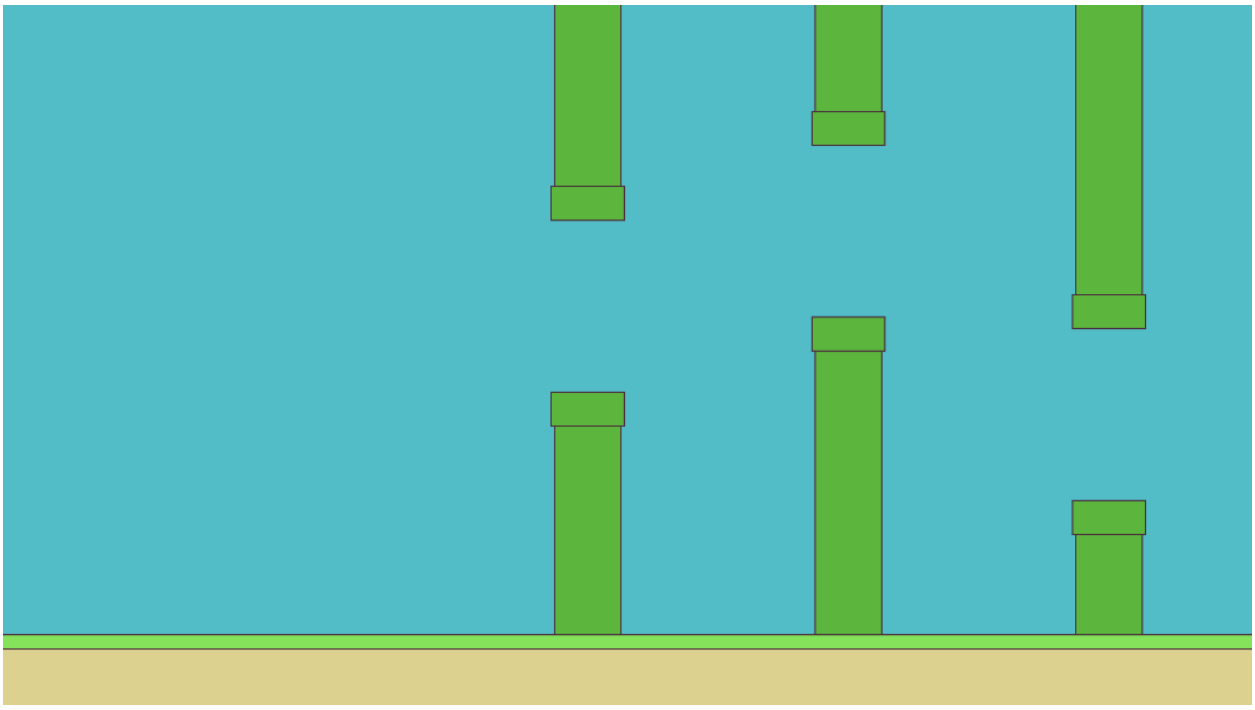
(continued from previous page)

```
    {
        // Ground expands to whole canvas width so position is middle of it,
        // But position Y is at bottom of the screen so it's full canvas_height minus␣
↪half of the ground thickness
        transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness * 0.5f};

        // Size X is full canvas but the height is defined in constants
        math::vec2f size{canvas_width, constants.ground_thickness};

        auto ground = geometry::blueprint_rectangle(registry, size, constants.ground_
↪color, pos);
        registry.assign<graphics::layer<3>>(ground);
        tag_game_scene(registry, ground);
    }
}
```

Let's call it inside the `game_scene` constructor.

```
game_scene(entt::registry &registry) noexcept : base_scene(registry) {
    // Set the constants that will be used in the program
    registry.set<flappy_bird_constants>();

    // Create everything
    create_background(registry);
    init_dynamic_objects(registry);
}
```

Now we have a pretty background, at least as pretty as it can get with three rectangles!



Step 3 is complete, full code below.

```
#include <random>
#include <antara/gaming/graphics/component.layer.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/graphics/component.sprite.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Constants
struct flappy_bird_constants {
    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};

    // Background
    const float ground_thickness{100.0f};
    const float grass_thickness{20.0f};
    const graphics::color background_color{82, 189, 199};
    const graphics::color ground_color{220, 209, 143};
    const graphics::color grass_color{132, 227, 90};
    const graphics::outline_color grass_outline_color{2.0f, graphics::color{76, 47,␣
↪61}};
};

// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}

// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
    entt::entity cap{entt::null};
```

(continues on next page)

(continued from previous page)

```
    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};

// Column is made of two pipes
struct column {
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
        top_pipe.destroy(registry);
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
};

// Logic functions
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }

    // Returns a random gap start position Y
    float get_random_gap_start_pos(const entt::registry &registry) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        float top_limit = canvas_height * constants.column_min;
        float bottom_limit = canvas_height * constants.column_max - constants.gap_
↪height;

        return random_float(top_limit, bottom_limit);
    }
}

// Factory functions
namespace {
    // Factory for pipes, requires to know if it's a top one, position x of the␣
↪column, and the gap starting position Y
    pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_
↪start_pos_y) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();
```

(continues on next page)

(continued from previous page)

```
        // PIPE BODY
        // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the␣
↪canvas
        transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

        // Size X is the column thickness,
        // Size Y is the important part.
        // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
        //  So half size should be gap_start_pos_y since center of the rectangle is␣
↪at 0.
        // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y +␣
↪gap_height
        //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
        // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
        math::vec2f body_size{constants.column_thickness,
                              is_top ?
                              gap_start_pos_y * 2.0f :
                              (canvas_height - (gap_start_pos_y + constants.gap_
↪height)) * 2.0f};

        auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                                  constants.pipe_outline_color);

        // PIPE CAP
        // Let's prepare the pipe cap
        // Size of the cap is defined in constants
        math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_
↪width, constants.pipe_cap_height};

        // Position, X is same as the body. Bottom of the cap is aligned with bottom␣
↪of the body,
        // or start of the gap, we will use start of the gap here, minus half of the␣
↪cap height
        transform::position_2d cap_pos{body_pos.x(),
                                       is_top ?
                                       gap_start_pos_y - constants.pipe_cap_height *␣
↪0.5f :
                                       gap_start_pos_y + constants.gap_height +␣
↪constants.pipe_cap_height * 0.5f
        };

        // Construct the cap
        auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_
↪color, cap_pos,
                                                 constants.pipe_outline_color);

        // Set layers, cap should be in front of body
        registry.assign<graphics::layer<4>>(cap);
        registry.assign<graphics::layer<3>>(body);
        tag_game_scene(registry, cap, true);
        tag_game_scene(registry, body, true);

        // Construct a pipe with body and cap and return it
        return {body, cap};
```

(continues on next page)

(continued from previous page)

```
    }

    // Factory to create single column
    void create_column(entt::registry &registry, float pos_x) noexcept {
        // Create a fresh entity for a new column
        auto entity_column = registry.create();

        // Get a random gap start position Y, between pipes
        float gap_start_pos_y = get_random_gap_start_pos(registry);

        // Create pipes, is_top variable is false for bottom one
        auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
        auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

        // Make a column from these two pipes and mark it as "column"
        registry.assign<column>(entity_column, top_pipe, bottom_pipe);
        registry.assign<entt::tag<"column"_hs>>(entity_column);
        tag_game_scene(registry, entity_column, true);
    }

    // Factory for creating a Flappy Bird columns
    void create_columns(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Spawn columns out of the screen, out of the canvas
        const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

        // Create the columns
        for (std::size_t i = 0; i < constants.column_count; ++i) {
            // Horizontal position (X) increases for every column, keeping the␣
↪distance
            float pos_x = column_pos_offset + i * constants.column_distance;

            create_column(registry, pos_x);
        }
    }

    // Factory for creating a Flappy Bird background
    void create_background(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Create Sky
        {
            // Sky is whole canvas so position is middle of it
            transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

            // And the size is full canvas
            math::vec2f size{canvas_width, canvas_height};

            auto sky = geometry::blueprint_rectangle(registry, size, constants.
↪background_color, pos);
            registry.assign<graphics::layer<1>>(sky);
```

(continues on next page)

(continued from previous page)

```
            tag_game_scene(registry, sky);
        }

        // Create Grass
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at top of the ground, so it's canvas height minus␣
↪ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness};

            // Size X is full canvas but the height is defined in constants
            // We also make it a bit longer by adding the thickness of the outline to␣
↪hide the outline at sides
            math::vec2f size{canvas_width + constants.grass_outline_color.thickness *␣
↪2.0f, constants.grass_thickness};

            auto grass = geometry::blueprint_rectangle(registry, size, constants.
↪grass_color, pos,
                                                       constants.grass_outline_color);
            registry.assign<graphics::layer<3>>(grass);
            tag_game_scene(registry, grass);
        }

        // Create Ground
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at bottom of the screen so it's full canvas_height␣
↪minus half of the ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness * 0.5f};

            // Size X is full canvas but the height is defined in constants
            math::vec2f size{canvas_width, constants.ground_thickness};

            auto ground = geometry::blueprint_rectangle(registry, size, constants.
↪ground_color, pos);
            registry.assign<graphics::layer<3>>(ground);
            tag_game_scene(registry, ground);
        }
    }
}

// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry) noexcept : base_scene(registry) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();

        // Create everything
        create_background(registry);
        init_dynamic_objects(registry);
    }

    // Scene name
    std::string scene_name() noexcept final {
```

(continues on next page)

(continued from previous page)

```
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
    }

    // Initialize dynamic objects, this function is called at start and resets
    void init_dynamic_objects(entt::registry &registry) {
        create_columns(registry);
    }
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_),␣
↪true);
    }
};

int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

Step 4: Move, destroy and respawn pipes

To create the illusion of movement, Flappy Bird will stay still and pipes will move to left. That way, we don't need to make a camera which follows Flappy Bird.

Let's define a constant into `flappy_bird_constants` named `scroll_speed`, which will be the movement speed of pipes moving left (towards Flappy bird).

```
const float scroll_speed{200.f};
```

Smooth movement requires a position update at every tick. So we need a `ecs::logic_update_system`, let's

call it `column_logic`.

```
// Column Logic System
class column_logic final : public ecs::logic_update_system<column_logic> {
public:
    explicit column_logic(entt::registry &registry) noexcept : system(registry) {
        disable();
    }
```

Next we make a `move_pipe` function, with a parameter of `struct pipe` reference. We'll also retrieve constants, to access the scroll speed.

```
// Move the pipe and return the x position
float move_pipe(entt::registry &registry, pipe &pipe) {
    // Retrieve constants
    const auto constants = registry.ctx<flappy_bird_constants>();
```

To move the pipe, first we need to know its current position. We retrieve the body position of the pipe (cap is also the same so body will be enough). Pipes move only along the X axis, horizontally. So we are only interested in their X position.

```
// Get current position of the pipe
auto pos = registry.get<transform::position_2d>(pipe.body);
```

Now we calculate the new position X, by adding `scroll_speed`, but we use `-` because a lower position value moves to the left side. So we're actually subtracting from the X position value to make the pipe move to left side. We also multiply `scroll_speed` with delta time `timer::time_step::get_fixed_delta_time()`, so the movement will be spread over time, and the frame changes will look smoother. `scroll_speed` is actually amount of pixels the object will move in `1 second`.

```
// Shift pos X to left by scroll_speed but multiplying with dt because we do this so␣
↪many times a second,
// Delta time makes sure that it's applying over time, so in one second it will move␣
↪scroll_speed pixels
auto new_pos_x = pos.x() - constants.scroll_speed * timer::time_step::get_fixed_delta_
↪time();
```

Update both body and cap positions by replacing entity's `transform::position_2d`.

```
// Set the new position value
registry.replace<transform::position_2d>(pipe.body, new_pos_x, pos.y());

// Set cap position too
auto cap_pos = registry.get<transform::position_2d>(pipe.cap);
registry.replace<transform::position_2d>(pipe.cap, new_pos_x, cap_pos.y());
```

And finally, return the new position (we will use this later).

```
// Return the info about if this pipe is out of the screen
return new_pos_x;
```

The completed function now looks like this:

```
// Move the pipe and return the x position
float move_pipe(entt::registry &registry, pipe &pipe) {
    // Retrieve constants
    const auto constants = registry.ctx<flappy_bird_constants>();
```

(continues on next page)

(continued from previous page)

```
    // Get current position of the pipe
    auto pos = registry.get<transform::position_2d>(pipe.body);

    // Shift pos X to left by scroll_speed but multiplying with dt because we do this␣
↪so many times a second,
    // Delta time makes sure that it's applying over time, so in one second it will␣
↪move scroll_speed pixels
    auto new_pos_x = pos.x() - constants.scroll_speed * timer::time_step::get_fixed_
↪delta_time();

    // Set the new position value
    registry.replace<transform::position_2d>(pipe.body, new_pos_x, pos.y());

    // Set cap position too
    auto cap_pos = registry.get<transform::position_2d>(pipe.cap);
    registry.replace<transform::position_2d>(pipe.cap, new_pos_x, cap_pos.y());

    // Return the info about if this pipe is out of the screen
    return new_pos_x;
}
```

In the update function, we will move all the columns every tick, and those exiting the screen on the left will be destroyed. A new column will be spawned coming in from the right side, the column furthest away from Flappy bird. Let's make a very basic function which will return the X position of the furthest pipe.

This basic function simply loops over all columns and check if the column's X position is higher than the previous maximum.

```
// Find the furthest pipe's position X
float furthest_pipe_position(entt::registry &registry) {
    float furthest = 0.f;

    for (auto entity : registry.view<column>()) {
        auto &col = registry.get<column>(entity);
        float x = entity_registry_.get<transform::position_2d>(col.top_pipe.body).x();
        if (x > furthest) furthest = x;
    }

    return furthest;
}
```

Now we can make the update function which will be called every single tick.

It will retrieve constants and all columns, then loop all the columns.

```
// Update, this will be called every tick
void update() noexcept final {
    auto &registry = entity_registry_;

    // Retrieve constants
    const auto constants = registry.ctx<flappy_bird_constants>();

    // Loop all columns
    for (auto entity : registry.view<column>()) {
```

Inside the loop, we get the `struct column` from the column `entt::entity`.

```
auto &col = registry.get<column>(entity);
```

Then call the `move_pipe` function twice, one for top pipe, one for the bottom one. They are at the same X position, so to know the column position, we save the return value of one of them into `column_pos_x`.

```
// Move pipes, and retrieve column position x
float column_pos_x = move_pipe(registry, col.top_pipe);
move_pipe(registry, col.bottom_pipe);
```

Now we know the column position. As we said before, we need to destroy it if it's out of the screen. Position of left side of the screen is `0`. To make sure column is out of the screen, we can use the `column_distance` value, but negative. For example, it will be `-400`. We can compare the column's X position against this value to know if it's outside the screen.

```
// If column is out of the screen
if (column_pos_x < -constants.column_distance) {
```

If it is, we destroy the column, then create a new column on the right using the `create_column` function. As the new column position, we use the `furthest_pipe_position` then add `column_distance` so it will spawn a little bit further than the last column.

```
// If column is out of the screen
if (column_pos_x < -constants.column_distance) {
    // Remove this column
    col.destroy(registry, entity);

    // Create a new column at far end
    create_column(registry, furthest_pipe_position(registry) + constants.column_
↪distance);
}
```

That's it - the complete `update` function looks like this:

```
// Update, this will be called every tick
void update() noexcept final {
    auto &registry = entity_registry_;

    // Retrieve constants
    const auto constants = registry.ctx<flappy_bird_constants>();

    // Loop all columns
    for (auto entity : registry.view<column>()) {
        auto &col = registry.get<column>(entity);

        // Move pipes, and retrieve column position x
        float column_pos_x = move_pipe(registry, col.top_pipe);
        move_pipe(registry, col.bottom_pipe);

        // If column is out of the screen
        if (column_pos_x < -constants.column_distance) {
            // Remove this column
            col.destroy(registry, entity);

            // Create a new column at far end
            create_column(registry, furthest_pipe_position(registry) + constants.
↪column_distance);
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Now we name this logic system, after the class.

```
// Name this system
REFL_AUTO (type(column_logic));
```

`column_logic` class is fully ready.

To create a logic system, we need to access `ecs::system_manager` inside the `game_scene`.

We add a member variable to store the reference inside the `game_scene`.

```
// System manager reference
ecs::system_manager &system_manager_;
```

Then add a parameter to the constructor which sets this reference.

```
game_scene(entt::registry &registry, ecs::system_manager &system_manager) noexcept :␣
↪base_scene(registry), system_manager_(system_manager) {
```

Now we will make a function which will create logic systems, inside we use the `system_manager_`.

```
// Create logic systems
void create_logic_systems() {
    system_manager_.create_system_rt<column_logic>();
}
```

And finally call this in the `init_dynamic_objects` function.

```
// Initialize dynamic objects, this function is called at start and resets
void init_dynamic_objects(entt::registry &registry) {
    create_columns(registry);

    // Create logic systems
    create_logic_systems();
}
```

Step 4 is now complete, here is the full code.

```
#include <random>
#include <antara/gaming/graphics/component.layer.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/graphics/component.sprite.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Constants
```

(continues on next page)

(continued from previous page)

```
struct flappy_bird_constants {
    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};
    const float scroll_speed{200.f};

    // Background
    const float ground_thickness{100.0f};
    const float grass_thickness{20.0f};
    const graphics::color background_color{82, 189, 199};
    const graphics::color ground_color{220, 209, 143};
    const graphics::color grass_color{132, 227, 90};
    const graphics::outline_color grass_outline_color{2.0f, graphics::color{76, 47,␣
↪61}};
};

// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}

// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
    entt::entity cap{entt::null};

    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};

// Column is made of two pipes
struct column {
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
```

(continues on next page)

(continued from previous page)

```
        top_pipe.destroy(registry);
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
};

// Logic functions
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }

    // Returns a random gap start position Y
    float get_random_gap_start_pos(const entt::registry &registry) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        float top_limit = canvas_height * constants.column_min;
        float bottom_limit = canvas_height * constants.column_max - constants.gap_
↪height;

        return random_float(top_limit, bottom_limit);
    }
}

// Factory functions
namespace {
    // Factory for pipes, requires to know if it's a top one, position x of the␣
↪column, and the gap starting position Y
    pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_
↪start_pos_y) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        // PIPE BODY
        // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the␣
↪canvas
        transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

        // Size X is the column thickness,
        // Size Y is the important part.
        // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
        //  So half size should be gap_start_pos_y since center of the rectangle is␣
↪at 0.
        // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y +␣
↪gap_height
        //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
        // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
```

(continues on next page)

(continued from previous page)

```
        math::vec2f body_size{constants.column_thickness,
                              is_top ?
                              gap_start_pos_y * 2.0f :
                              (canvas_height - (gap_start_pos_y + constants.gap_
↪height)) * 2.0f};

        auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                                  constants.pipe_outline_color);

        // PIPE CAP
        // Let's prepare the pipe cap
        // Size of the cap is defined in constants
        math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_
↪width, constants.pipe_cap_height};

        // Position, X is same as the body. Bottom of the cap is aligned with bottom␣
↪of the body,
        // or start of the gap, we will use start of the gap here, minus half of the␣
↪cap height
        transform::position_2d cap_pos{body_pos.x(),
                                       is_top ?
                                       gap_start_pos_y - constants.pipe_cap_height *␣
↪0.5f :
                                       gap_start_pos_y + constants.gap_height +␣
↪constants.pipe_cap_height * 0.5f
        };

        // Construct the cap
        auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_
↪color, cap_pos,
                                                 constants.pipe_outline_color);

        // Set layers, cap should be in front of body
        registry.assign<graphics::layer<4>>(cap);
        registry.assign<graphics::layer<3>>(body);
        tag_game_scene(registry, cap, true);
        tag_game_scene(registry, body, true);

        // Construct a pipe with body and cap and return it
        return {body, cap};
    }

    // Factory to create single column
    void create_column(entt::registry &registry, float pos_x) noexcept {
        // Create a fresh entity for a new column
        auto entity_column = registry.create();

        // Get a random gap start position Y, between pipes
        float gap_start_pos_y = get_random_gap_start_pos(registry);

        // Create pipes, is_top variable is false for bottom one
        auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
        auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

        // Make a column from these two pipes and mark it as "column"
        registry.assign<column>(entity_column, top_pipe, bottom_pipe);
```

(continues on next page)

(continued from previous page)

```
        registry.assign<entt::tag<"column"_hs>>(entity_column);
        tag_game_scene(registry, entity_column, true);
    }

    // Factory for creating a Flappy Bird columns
    void create_columns(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Spawn columns out of the screen, out of the canvas
        const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

        // Create the columns
        for (std::size_t i = 0; i < constants.column_count; ++i) {
            // Horizontal position (X) increases for every column, keeping the␣
↪distance
            float pos_x = column_pos_offset + i * constants.column_distance;

            create_column(registry, pos_x);
        }
    }

    // Factory for creating a Flappy Bird background
    void create_background(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Create Sky
        {
            // Sky is whole canvas so position is middle of it
            transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

            // And the size is full canvas
            math::vec2f size{canvas_width, canvas_height};

            auto sky = geometry::blueprint_rectangle(registry, size, constants.
↪background_color, pos);
            registry.assign<graphics::layer<1>>(sky);
            tag_game_scene(registry, sky);
        }

        // Create Grass
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at top of the ground, so it's canvas height minus␣
↪ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness};

            // Size X is full canvas but the height is defined in constants
            // We also make it a bit longer by adding the thickness of the outline to␣
↪hide the outline at sides
            math::vec2f size{canvas_width + constants.grass_outline_color.thickness *␣
↪2.0f, constants.grass_thickness};
```

(continues on next page)

(continued from previous page)

```
            auto grass = geometry::blueprint_rectangle(registry, size, constants.
↪grass_color, pos,
                                                       constants.grass_outline_color);
            registry.assign<graphics::layer<3>>(grass);
            tag_game_scene(registry, grass);
        }

        // Create Ground
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at bottom of the screen so it's full canvas_height␣
↪minus half of the ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness * 0.5f};

            // Size X is full canvas but the height is defined in constants
            math::vec2f size{canvas_width, constants.ground_thickness};

            auto ground = geometry::blueprint_rectangle(registry, size, constants.
↪ground_color, pos);
            registry.assign<graphics::layer<3>>(ground);
            tag_game_scene(registry, ground);
        }
    }
}

// Column Logic System
class column_logic final : public ecs::logic_update_system<column_logic> {
public:
    explicit column_logic(entt::registry &registry) noexcept : system(registry) {
        disable();
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Loop all columns
        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);

            // Move pipes, and retrieve column position x
            float column_pos_x = move_pipe(registry, col.top_pipe);
            move_pipe(registry, col.bottom_pipe);

            // If column is out of the screen
            if (column_pos_x < -constants.column_distance) {
                // Remove this column
                col.destroy(registry, entity);

                // Create a new column at far end
                create_column(registry, furthest_pipe_position(registry) + constants.
↪column_distance);
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }

private:
    // Find the furthest pipe's position X
    float furthest_pipe_position(entt::registry &registry) {
        float furthest = 0.f;

        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);
            float x = entity_registry_.get<transform::position_2d>(col.top_pipe.body).
↪x();
            if (x > furthest) furthest = x;
        }

        return furthest;
    }

    // Move the pipe and return the x position
    float move_pipe(entt::registry &registry, pipe &pipe) {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the pipe
        auto pos = registry.get<transform::position_2d>(pipe.body);

        // Shift pos X to left by scroll_speed but multiplying with dt because we do␣
↪this so many times a second,
        // Delta time makes sure that it's applying over time, so in one second it␣
↪will move scroll_speed pixels
        auto new_pos_x = pos.x() - constants.scroll_speed * timer::time_step::get_
↪fixed_delta_time();

        // Set the new position value
        registry.replace<transform::position_2d>(pipe.body, new_pos_x, pos.y());

        // Set cap position too
        auto cap_pos = registry.get<transform::position_2d>(pipe.cap);
        registry.replace<transform::position_2d>(pipe.cap, new_pos_x, cap_pos.y());

        // Return the info about if this pipe is out of the screen
        return new_pos_x;
    }
};

// Name this system
REFL_AUTO (type(column_logic));

// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry, ecs::system_manager &system_manager)␣
↪noexcept : base_scene(registry), system_manager_(system_manager) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();
```

(continues on next page)

(continued from previous page)

```
        // Create everything
        create_background(registry);
        init_dynamic_objects(registry);
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
    }

    // Initialize dynamic objects, this function is called at start and resets
    void init_dynamic_objects(entt::registry &registry) {
        create_columns(registry);

        // Create logic systems
        create_logic_systems();
    }

    // Create logic systems
    void create_logic_systems() {
        system_manager_.create_system_rt<column_logic>();
    }

    // System manager reference
    ecs::system_manager &system_manager_;
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_,␣
↪system_manager_), true);
    }
};

int main() {
    // Declare the world
```

(continues on next page)

(continued from previous page)

```
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

Step 5: Creation of Flappy Bird

Now we will create the Flappy Bird. Instead of using a rectangle as a character, we will use an image file. This is called a `character sprite`. We will call Flappy Bird "player" from now on.

We need two constants, one for the player position, and another one for the image file name.

```
struct flappy_bird_constants {
    // Player
    const std::string player_image_name{"player.png"};
    const float player_pos_x{400.0f};
```

Let's make a `create_player` function which will construct the player entity and return it.

We retrieve the constants as always.

```
// Factory for creating the player
entt::entity create_player(entt::registry &registry) {
    // Retrieve constants
    const auto[_, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.size;
    const auto constants = registry.ctx<flappy_bird_constants>();
```

Then we use the `graphics::blueprint_sprite` which is really easy to use. It requires two parameters, `graphics::sprite` and `transform::position_2d`. `graphics::sprite` gets the image path, and `transform::position_2d` gets the `player_pos_x` constant as X position, half of the canvas height as Y position.

```
auto entity = graphics::blueprint_sprite(registry,
                                         graphics::sprite{constants.player_image_
↪name.c_str()},
                                         transform::position_2d{constants.player_
↪pos_x, canvas_height * 0.5f});
```

We assign `layer<5>` for draw order, tag `player`, `game_scene` and `dynamic`. Then return the entity.

```
registry.assign<antara::gaming::graphics::layer<5>>(entity);
registry.assign<entt::tag<"player"_hs>>(entity);
tag_game_scene(registry, entity, true);

return entity;
```

The completed `create_player` function looks like this:

```
// Factory for creating the player
entt::entity create_player(entt::registry &registry) {
    // Retrieve constants
    const auto[_, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.size;
    const auto constants = registry.ctx<flappy_bird_constants>();
```

(continues on next page)

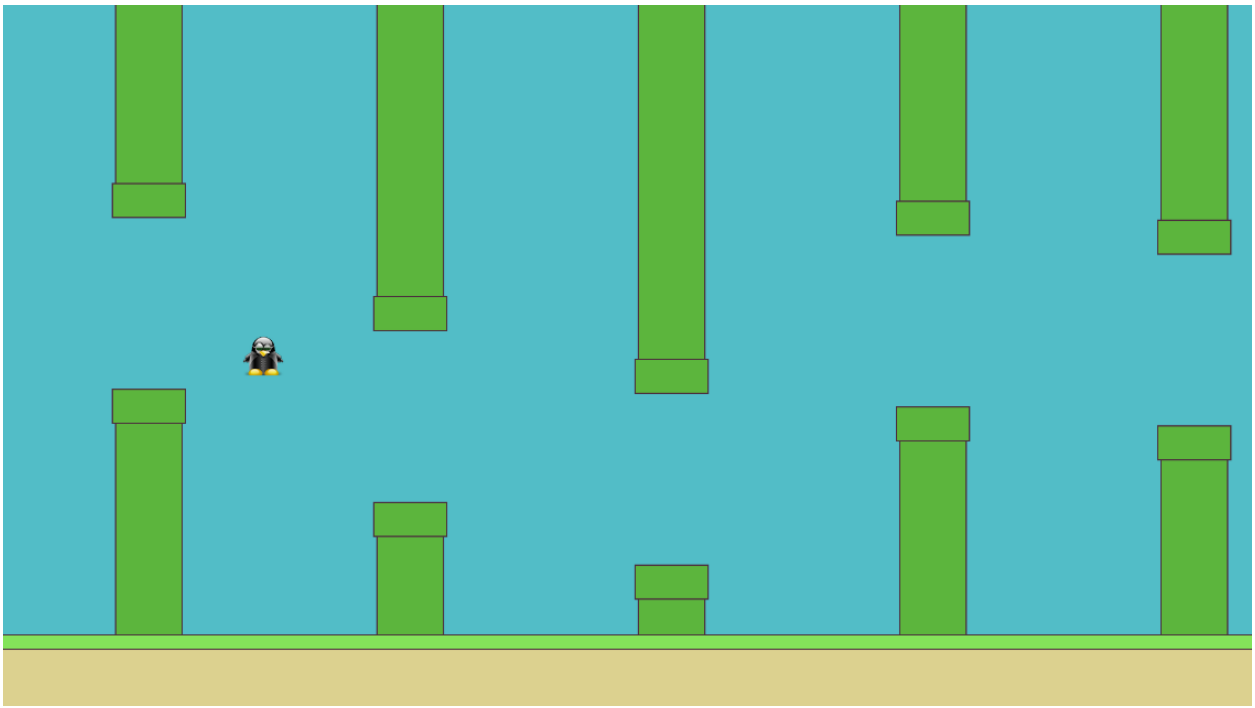
(continued from previous page)

```
    auto entity = graphics::blueprint_sprite(registry,
                                             graphics::sprite{constants.player_image_
↪name.c_str()},
                                             transform::position_2d{constants.player_
↪pos_x, canvas_height * 0.5f});
    registry.assign<antara::gaming::graphics::layer<5>>(entity);
    registry.assign<entt::tag<"player"_hs>>(entity);
    tag_game_scene(registry, entity, true);

    return entity;
}
```

Finally we call this function inside `init_dynamic_objects`.

```
// Initialize dynamic objects, this function is called at start and resets
void init_dynamic_objects(entt::registry &registry) {
    create_columns(registry);

    // Create player
    create_player(registry);

    // Create logic systems
    create_logic_systems();
}
```

Now you should be able to see the character and moving pipes.



Step 5 is complete, here is the full code.

```
#include <random>
#include <antara/gaming/graphics/component.layer.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/graphics/component.sprite.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Constants
struct flappy_bird_constants {
    // Player
    const std::string player_image_name{"player.png"};
    const float player_pos_x{400.0f};

    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};
    const float scroll_speed{200.f};

    // Background
    const float ground_thickness{100.0f};
    const float grass_thickness{20.0f};
    const graphics::color background_color{82, 189, 199};
    const graphics::color ground_color{220, 209, 143};
    const graphics::color grass_color{132, 227, 90};
    const graphics::outline_color grass_outline_color{2.0f, graphics::color{76, 47,␣
↪61}};
};

// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}

// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
```

(continues on next page)

(continued from previous page)

```
    entt::entity cap{entt::null};

    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};

// Column is made of two pipes
struct column {
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
        top_pipe.destroy(registry);
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
};

// Logic functions
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }

    // Returns a random gap start position Y
    float get_random_gap_start_pos(const entt::registry &registry) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        float top_limit = canvas_height * constants.column_min;
        float bottom_limit = canvas_height * constants.column_max - constants.gap_
↪height;

        return random_float(top_limit, bottom_limit);
    }
}

// Factory functions
namespace {
    // Factory for pipes, requires to know if it's a top one, position x of the␣
↪column, and the gap starting position Y
    pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_
↪start_pos_y) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
```

(continues on next page)

(continued from previous page)

```
        const auto constants = registry.ctx<flappy_bird_constants>();

        // PIPE BODY
        // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the␣
↪canvas
        transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

        // Size X is the column thickness,
        // Size Y is the important part.
        // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
        //  So half size should be gap_start_pos_y since center of the rectangle is␣
↪at 0.
        // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y +␣
↪gap_height
        //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
        // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
        math::vec2f body_size{constants.column_thickness,
                              is_top ?
                              gap_start_pos_y * 2.0f :
                              (canvas_height - (gap_start_pos_y + constants.gap_
↪height)) * 2.0f};

        auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                                  constants.pipe_outline_color);

        // PIPE CAP
        // Let's prepare the pipe cap
        // Size of the cap is defined in constants
        math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_
↪width, constants.pipe_cap_height};

        // Position, X is same as the body. Bottom of the cap is aligned with bottom␣
↪of the body,
        // or start of the gap, we will use start of the gap here, minus half of the␣
↪cap height
        transform::position_2d cap_pos{body_pos.x(),
                                       is_top ?
                                       gap_start_pos_y - constants.pipe_cap_height *␣
↪0.5f :
                                       gap_start_pos_y + constants.gap_height +␣
↪constants.pipe_cap_height * 0.5f
        };

        // Construct the cap
        auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_
↪color, cap_pos,
                                                 constants.pipe_outline_color);

        // Set layers, cap should be in front of body
        registry.assign<graphics::layer<4>>(cap);
        registry.assign<graphics::layer<3>>(body);
        tag_game_scene(registry, cap, true);
        tag_game_scene(registry, body, true);

        // Construct a pipe with body and cap and return it
```

(continues on next page)

(continued from previous page)

```
        return {body, cap};
    }

    // Factory to create single column
    void create_column(entt::registry &registry, float pos_x) noexcept {
        // Create a fresh entity for a new column
        auto entity_column = registry.create();

        // Get a random gap start position Y, between pipes
        float gap_start_pos_y = get_random_gap_start_pos(registry);

        // Create pipes, is_top variable is false for bottom one
        auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
        auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

        // Make a column from these two pipes and mark it as "column"
        registry.assign<column>(entity_column, top_pipe, bottom_pipe);
        registry.assign<entt::tag<"column"_hs>>(entity_column);
        tag_game_scene(registry, entity_column, true);
    }

    // Factory for creating a Flappy Bird columns
    void create_columns(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Spawn columns out of the screen, out of the canvas
        const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

        // Create the columns
        for (std::size_t i = 0; i < constants.column_count; ++i) {
            // Horizontal position (X) increases for every column, keeping the␣
↪distance
            float pos_x = column_pos_offset + i * constants.column_distance;

            create_column(registry, pos_x);
        }
    }

    // Factory for creating a Flappy Bird background
    void create_background(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Create Sky
        {
            // Sky is whole canvas so position is middle of it
            transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

            // And the size is full canvas
            math::vec2f size{canvas_width, canvas_height};

            auto sky = geometry::blueprint_rectangle(registry, size, constants.
↪background_color, pos);
```

(continues on next page)

(continued from previous page)

```
            registry.assign<graphics::layer<1>>(sky);
            tag_game_scene(registry, sky);
        }

        // Create Grass
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at top of the ground, so it's canvas height minus␣
↪ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness};

            // Size X is full canvas but the height is defined in constants
            // We also make it a bit longer by adding the thickness of the outline to␣
↪hide the outline at sides
            math::vec2f size{canvas_width + constants.grass_outline_color.thickness *␣
↪2.0f, constants.grass_thickness};

            auto grass = geometry::blueprint_rectangle(registry, size, constants.
↪grass_color, pos,
                                                       constants.grass_outline_color);
            registry.assign<graphics::layer<3>>(grass);
            tag_game_scene(registry, grass);
        }

        // Create Ground
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at bottom of the screen so it's full canvas_height␣
↪minus half of the ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness * 0.5f};

            // Size X is full canvas but the height is defined in constants
            math::vec2f size{canvas_width, constants.ground_thickness};

            auto ground = geometry::blueprint_rectangle(registry, size, constants.
↪ground_color, pos);
            registry.assign<graphics::layer<3>>(ground);
            tag_game_scene(registry, ground);
        }
    }

    // Factory for creating the player
    entt::entity create_player(entt::registry &registry) {
        // Retrieve constants
        const auto[_, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.
↪size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        auto entity = graphics::blueprint_sprite(registry,
                                                 graphics::sprite{constants.player_
↪image_name.c_str()},
                                                 transform::position_2d{constants.
↪player_pos_x, canvas_height * 0.5f});
        registry.assign<antara::gaming::graphics::layer<5>>(entity);
        registry.assign<entt::tag<"player"_hs>>(entity);
```

(continues on next page)

(continued from previous page)

```
        tag_game_scene(registry, entity, true);

        return entity;
    }
}

// Column Logic System
class column_logic final : public ecs::logic_update_system<column_logic> {
public:
    explicit column_logic(entt::registry &registry) noexcept : system(registry) {
        disable();
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Loop all columns
        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);

            // Move pipes, and retrieve column position x
            float column_pos_x = move_pipe(registry, col.top_pipe);
            move_pipe(registry, col.bottom_pipe);

            // If column is out of the screen
            if (column_pos_x < -constants.column_distance) {
                // Remove this column
                col.destroy(registry, entity);

                // Create a new column at far end
                create_column(registry, furthest_pipe_position(registry) + constants.
↪column_distance);
            }
        }
    }

private:
    // Find the furthest pipe's position X
    float furthest_pipe_position(entt::registry &registry) {
        float furthest = 0.f;

        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);
            float x = entity_registry_.get<transform::position_2d>(col.top_pipe.body).
↪x();
            if (x > furthest) furthest = x;
        }

        return furthest;
    }

    // Move the pipe and return the x position
    float move_pipe(entt::registry &registry, pipe &pipe) {
```

(continues on next page)

(continued from previous page)

```
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the pipe
        auto pos = registry.get<transform::position_2d>(pipe.body);

        // Shift pos X to left by scroll_speed but multiplying with dt because we do␣
↪this so many times a second,
        // Delta time makes sure that it's applying over time, so in one second it␣
↪will move scroll_speed pixels
        auto new_pos_x = pos.x() - constants.scroll_speed * timer::time_step::get_
↪fixed_delta_time();

        // Set the new position value
        registry.replace<transform::position_2d>(pipe.body, new_pos_x, pos.y());

        // Set cap position too
        auto cap_pos = registry.get<transform::position_2d>(pipe.cap);
        registry.replace<transform::position_2d>(pipe.cap, new_pos_x, cap_pos.y());

        // Return the info about if this pipe is out of the screen
        return new_pos_x;
    }
};

// Name this system
REFL_AUTO (type(column_logic));

// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry, ecs::system_manager &system_manager)␣
↪noexcept : base_scene(registry), system_manager_(system_manager) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();

        // Create everything
        create_background(registry);
        init_dynamic_objects(registry);
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
    }

    // Initialize dynamic objects, this function is called at start and resets
    void init_dynamic_objects(entt::registry &registry) {
        create_columns(registry);

        // Create player
        create_player(registry);
```

(continues on next page)

(continued from previous page)

```
        // Create logic systems
        create_logic_systems();
    }

    // Create logic systems
    void create_logic_systems() {
        system_manager_.create_system_rt<column_logic>();
    }

    // System manager reference
    ecs::system_manager &system_manager_;
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_,␣
↪system_manager_), true);
    }
};

int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

Step 6: Player input and character physics

In this step, we will read user input and apply physics to the character.

We need to include two headers for input, `<antara/gaming/input/virtual.hpp>` and `<antara/gaming/ecs/virtual.input.system.hpp>`.

We will also need some constants for physics. `gravity` is the force which will pull Flappy Bird down. `jump_force` will be the force which will be applied instantly when user presses the jump button. `rotate_speed` is for the rotating animation, and `max_angle` is the rotation limit.

```
const float gravity{2000.f};
const float jump_force{650.f};
const float rotate_speed{100.f};
const float max_angle{60.f};
```

Let's initialize the virtual input system and add a `jump` action. Keyboard keys will be: `space`, `w`, `up`; mouse buttons will be `left` and `right`.

```
// Create virtual input system
system_manager_.create_system<ecs::virtual_input_system>();

// Define the buttons for the jump action
input::virtual_input::create("jump",
                             {input::key::space, input::key::w, input::key::up},
                             {input::mouse_button::left, input::mouse_
↪button::right});
```

Now we will make another `ecs::logic_update_system` like `column_logic`, but this time for player.

```
// Player Logic System
class player_logic final : public ecs::logic_update_system<player_logic> {
public:
    player_logic(entt::registry &registry, entt::entity player_) noexcept :␣
↪system(registry), player_(player_) {
        disable();
    }
```

As you see in the constructor, we will keep the player entity as a member. Also we want a 2D vector for movement speed, `math::vec2f`.

```
private:
    entt::entity player_;
    math::vec2f movement_speed_{0.f, 0.f};
```

Now we can make the update function which will be called every tick.

```
// Update, this will be called every tick
void update() noexcept final {
    auto &registry = entity_registry_;

    // Retrieve constants
    const auto constants = registry.ctx<flappy_bird_constants>();

    // Get current position of the player
    auto pos = registry.get<transform::position_2d>(player_);
```

Adding gravity is really easy. As you know, gravity is acceleration, so instead of adding it to the position, we add it to the movement speed. We multiply it with delta time to spread it over time.

Updating Y of `movement_speed_` with Y plus gravity.

```
// Add gravity to movement speed, multiply with delta time to apply it over time
movement_speed_.set_y(movement_speed_.y() + constants.gravity * timer::time_step::get_
↪fixed_delta_time());
```

For jump, we check if jump button is tapped.

```
// Check if jump key is tapped
bool jump_key_tapped = input::virtual_input::is_tapped("jump");
```

If jump is tapped, we set Y of `movement_speed_` as negative `jump_force`. Negative because low values are up and high values are down, negative is being up.

Here we do a direct set instead of adding it on top of the previous value because we don't want player to spam jump button and infinitely speed up. Another problem could be movement speed Y is 900 and player presses jump button, adding -650, player still will have 250 movement speed Y, which is going down. We definitely do not want this, that's why we set instead of add.

```
// If jump is tapped, jump by adding jump force to the movement speed Y
if (jump_key_tapped) movement_speed_.set_y(-constants.jump_force);
```

Movement speed is ready. Now we move the position with the movement speed. Multiplying it with delta time as always to spread it over time.

```
// Add movement speed to position to make the character move, but apply over time␣
↪with delta time
pos += movement_speed_ * timer::time_step::get_fixed_delta_time();
```

Player can keep jumping and go out of the screen, so we need to limit the character position to stay inside.

If position Y is equal or lower than zero, we reset both position and speed Y to 0. That will keep the player inside no matter how many times jump is pressed.

```
// Do not let player to go out of the screen to top
if (pos.y() <= 0.f) {
    pos.set_y(0.f);
    movement_speed_.set_y(0.f);
}
```

Then set the modified position to the player entity.

```
// Set the new position value
registry.replace<transform::position_2d>(player_, pos);
```

Now player can jump, falls down with gravity, and is forced to stay inside the screen.

```
// Set the new position value
registry.replace<transform::position_2d>(player_, pos);
```

So far so good, but we still need to apply rotation to Flappy Bird, so he is looking down when falling.

To do this, retrieve the properties of the player, then add `rotate_speed` to the `props.rotation`, also apply delta time.

```
// ROTATION
// Retrieve props of the player
auto &props = registry.get<transform::properties>(player_);

// Increase the rotation a little by applying delta time
float new_rotation = props.rotation + constants.rotate_speed * timer::time_step::get_
↪fixed_delta_time();
```

When player jumps, we need to reset the rotation so character will be straight again before rotating back down. Also we don't want character to rotate forever, so we'll apply a `max_angle` limit.

```
// If jump button is tapped, reset rotation,
// If rotation is higher than the max angle, set it to max angle
if (jump_key_tapped)
    new_rotation = 0.f;
else if (props.rotation > constants.max_angle)
    new_rotation = constants.max_angle;
```

Finally, set the `transform::properties` to apply the rotation change.

```
// Set the properties
registry.replace<transform::properties>(player_, transform::properties{.rotation =␣
↪new_rotation});
```

Update function is complete, this is how it looks like:

```
// Update, this will be called every tick
void update() noexcept final {
    auto &registry = entity_registry_;

    // Retrieve constants
    const auto constants = registry.ctx<flappy_bird_constants>();

    // Get current position of the player
    auto pos = registry.get<transform::position_2d>(player_);

    // Add gravity to movement speed, multiply with delta time to apply it over time
    movement_speed_.set_y(movement_speed_.y() + constants.gravity * timer::time_
↪step::get_fixed_delta_time());

    // Check if jump key is tapped
    bool jump_key_tapped = input::virtual_input::is_tapped("jump");

    // If jump is tapped, jump by adding jump force to the movement speed Y
    if (jump_key_tapped) movement_speed_.set_y(-constants.jump_force);

    // Add movement speed to position to make the character move, but apply over time␣
↪with delta time
    pos += movement_speed_ * timer::time_step::get_fixed_delta_time();

    // Do not let player to go out of the screen to top
    if (pos.y() <= 0.f) {
        pos.set_y(0.f);
        movement_speed_.set_y(0.f);
    }

    // Set the new position value
    registry.replace<transform::position_2d>(player_, pos);

    // ROTATION
    // Retrieve props of the player
    auto &props = registry.get<transform::properties>(player_);

    // Increase the rotation a little by applying delta time
    float new_rotation = props.rotation + constants.rotate_speed * timer::time_
↪step::get_fixed_delta_time();

    // If jump button is tapped, reset rotation,
```

(continues on next page)

(continued from previous page)

```
    // If rotation is higher than the max angle, set it to max angle
    if (jump_key_tapped)
        new_rotation = 0.f;
    else if (props.rotation > constants.max_angle)
        new_rotation = constants.max_angle;

    // Set the properties
    registry.replace<transform::properties>(player_, transform::properties{.rotation␣
↪= new_rotation});
}
```

Now we'll name this logic system, after the class.

```
// Name this system
REFL_AUTO (type(player_logic));
```

`player_logic` is now ready. Let's use it in `game_scene`.

We made a function earlier called `create_logic_systems`, now we will create `player_logic` in it. Though `player_logic` requires `player` entity as argument. Modify the function like as below:

```
// Create logic systems
void create_logic_systems(entt::entity player) {
    system_manager_.create_system_rt<column_logic>();
    system_manager_.create_system_rt<player_logic>(player);
}
```

When we launch the game, we want physics in a paused state so we don't start the game until we press the jump button.

To do so, we'll make two functions which enable and disable both logic functions we made.

```
// Pause physics
void pause_physics() {
    system_manager_.disable_systems<column_logic, player_logic>();
}

// Resume physics
void resume_physics() {
    system_manager_.enable_systems<column_logic, player_logic>();
}
```

We'll use a boolean to indicate if player started playing.

```
// States
bool started_playing_{false};
```

And a function which resets this state value.

```
// Reset state values
void reset_state_variables() {
    started_playing_ = false;
}
```

In the `init_dynamic_objects` function, we feed `player` entity to the `create_logic_systems` function, pause physics, and reset state variables.

```
// Initialize dynamic objects, this function is called at start and resets
void init_dynamic_objects(entt::registry &registry) {
    create_columns(registry);

    // Create player
    auto player = create_player(registry);

    // Create logic systems
    create_logic_systems(player);

    // Reset state variables
    reset_state_variables();
}
```

Final thing we need to do is, to check for a jump button press which will start the game. We do this check only if player hasn't already started playing.

```
// Check if start game is requested at the pause state
void check_start_game_request() {
    // If game is not started yet and jump key is tapped
    if (!started_playing_ && input::virtual_input::is_tapped("jump")) {
        // Game starts, player started playing
        started_playing_ = true;
        resume_physics();
    }
}
```

Then call this function in the update function which gets called every tick.

```
// Update the game every tick
void update() noexcept final {
    // Check if player requested to start the game
    check_start_game_request();
}
```

Step 6 is complete, here is the full code.

```
#include <random>
#include <antara/gaming/ecs/virtual.input.system.hpp>
#include <antara/gaming/graphics/component.layer.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/graphics/component.sprite.hpp>
#include <antara/gaming/input/virtual.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Constants
struct flappy_bird_constants {
    // Player
```

(continues on next page)

(continued from previous page)

```
    const std::string player_image_name{"player.png"};
    const float player_pos_x{400.0f};
    const float gravity{2000.f};
    const float jump_force{650.f};
    const float rotate_speed{100.f};
    const float max_angle{60.f};

    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};
    const float scroll_speed{200.f};

    // Background
    const float ground_thickness{100.0f};
    const float grass_thickness{20.0f};
    const graphics::color background_color{82, 189, 199};
    const graphics::color ground_color{220, 209, 143};
    const graphics::color grass_color{132, 227, 90};
    const graphics::outline_color grass_outline_color{2.0f, graphics::color{76, 47,␣
↪61}};
};

// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}

// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
    entt::entity cap{entt::null};

    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};

// Column is made of two pipes
struct column {
```

(continues on next page)

(continued from previous page)

```
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
        top_pipe.destroy(registry);
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
};

// Logic functions
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }

    // Returns a random gap start position Y
    float get_random_gap_start_pos(const entt::registry &registry) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        float top_limit = canvas_height * constants.column_min;
        float bottom_limit = canvas_height * constants.column_max - constants.gap_
↪height;

        return random_float(top_limit, bottom_limit);
    }
}

// Factory functions
namespace {
    // Factory for pipes, requires to know if it's a top one, position x of the␣
↪column, and the gap starting position Y
    pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_
↪start_pos_y) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        // PIPE BODY
        // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the␣
↪canvas
        transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

        // Size X is the column thickness,
        // Size Y is the important part.
        // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
```

(continues on next page)

(continued from previous page)

```
        //  So half size should be gap_start_pos_y since center of the rectangle is␣
↪at 0.
        // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y +␣
↪gap_height
        //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
        // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
        math::vec2f body_size{constants.column_thickness,
                              is_top ?
                              gap_start_pos_y * 2.0f :
                              (canvas_height - (gap_start_pos_y + constants.gap_
↪height)) * 2.0f};

        auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                                  constants.pipe_outline_color);

        // PIPE CAP
        // Let's prepare the pipe cap
        // Size of the cap is defined in constants
        math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_
↪width, constants.pipe_cap_height};

        // Position, X is same as the body. Bottom of the cap is aligned with bottom␣
↪of the body,
        // or start of the gap, we will use start of the gap here, minus half of the␣
↪cap height
        transform::position_2d cap_pos{body_pos.x(),
                                       is_top ?
                                       gap_start_pos_y - constants.pipe_cap_height *␣
↪0.5f :
                                       gap_start_pos_y + constants.gap_height +␣
↪constants.pipe_cap_height * 0.5f
        };

        // Construct the cap
        auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_
↪color, cap_pos,
                                                 constants.pipe_outline_color);

        // Set layers, cap should be in front of body
        registry.assign<graphics::layer<4>>(cap);
        registry.assign<graphics::layer<3>>(body);
        tag_game_scene(registry, cap, true);
        tag_game_scene(registry, body, true);

        // Construct a pipe with body and cap and return it
        return {body, cap};
    }

    // Factory to create single column
    void create_column(entt::registry &registry, float pos_x) noexcept {
        // Create a fresh entity for a new column
        auto entity_column = registry.create();

        // Get a random gap start position Y, between pipes
        float gap_start_pos_y = get_random_gap_start_pos(registry);
```

(continues on next page)

(continued from previous page)

```
        // Create pipes, is_top variable is false for bottom one
        auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
        auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

        // Make a column from these two pipes and mark it as "column"
        registry.assign<column>(entity_column, top_pipe, bottom_pipe);
        registry.assign<entt::tag<"column"_hs>>(entity_column);
        tag_game_scene(registry, entity_column, true);
    }

    // Factory for creating a Flappy Bird columns
    void create_columns(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Spawn columns out of the screen, out of the canvas
        const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

        // Create the columns
        for (std::size_t i = 0; i < constants.column_count; ++i) {
            // Horizontal position (X) increases for every column, keeping the␣
↪distance
            float pos_x = column_pos_offset + i * constants.column_distance;

            create_column(registry, pos_x);
        }
    }

    // Factory for creating a Flappy Bird background
    void create_background(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Create Sky
        {
            // Sky is whole canvas so position is middle of it
            transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

            // And the size is full canvas
            math::vec2f size{canvas_width, canvas_height};

            auto sky = geometry::blueprint_rectangle(registry, size, constants.
↪background_color, pos);
            registry.assign<graphics::layer<1>>(sky);
            tag_game_scene(registry, sky);
        }

        // Create Grass
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at top of the ground, so it's canvas height minus␣
↪ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness};
```

(continues on next page)

(continued from previous page)

```
            // Size X is full canvas but the height is defined in constants
            // We also make it a bit longer by adding the thickness of the outline to␣
↪hide the outline at sides
            math::vec2f size{canvas_width + constants.grass_outline_color.thickness *␣
↪2.0f, constants.grass_thickness};

            auto grass = geometry::blueprint_rectangle(registry, size, constants.
↪grass_color, pos,
                                                       constants.grass_outline_color);
            registry.assign<graphics::layer<3>>(grass);
            tag_game_scene(registry, grass);
        }

        // Create Ground
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at bottom of the screen so it's full canvas_height␣
↪minus half of the ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness * 0.5f};

            // Size X is full canvas but the height is defined in constants
            math::vec2f size{canvas_width, constants.ground_thickness};

            auto ground = geometry::blueprint_rectangle(registry, size, constants.
↪ground_color, pos);
            registry.assign<graphics::layer<3>>(ground);
            tag_game_scene(registry, ground);
        }
    }

    // Factory for creating the player
    entt::entity create_player(entt::registry &registry) {
        // Retrieve constants
        const auto[_, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.
↪size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        auto entity = graphics::blueprint_sprite(registry,
                                                 graphics::sprite{constants.player_
↪image_name.c_str()},
                                                 transform::position_2d{constants.
↪player_pos_x, canvas_height * 0.5f});
        registry.assign<antara::gaming::graphics::layer<5>>(entity);
        registry.assign<entt::tag<"player"_hs>>(entity);
        tag_game_scene(registry, entity, true);

        return entity;
    }
}

// Column Logic System
class column_logic final : public ecs::logic_update_system<column_logic> {
public:
    explicit column_logic(entt::registry &registry) noexcept : system(registry) {
        disable();
```

(continues on next page)

(continued from previous page)

```
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Loop all columns
        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);

            // Move pipes, and retrieve column position x
            float column_pos_x = move_pipe(registry, col.top_pipe);
            move_pipe(registry, col.bottom_pipe);

            // If column is out of the screen
            if (column_pos_x < -constants.column_distance) {
                // Remove this column
                col.destroy(registry, entity);

                // Create a new column at far end
                create_column(registry, furthest_pipe_position(registry) + constants.
↪column_distance);
            }
        }
    }

private:
    // Find the furthest pipe's position X
    float furthest_pipe_position(entt::registry &registry) {
        float furthest = 0.f;

        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);
            float x = entity_registry_.get<transform::position_2d>(col.top_pipe.body).
↪x();
            if (x > furthest) furthest = x;
        }

        return furthest;
    }

    // Move the pipe and return the x position
    float move_pipe(entt::registry &registry, pipe &pipe) {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the pipe
        auto pos = registry.get<transform::position_2d>(pipe.body);

        // Shift pos X to left by scroll_speed but multiplying with dt because we do␣
↪this so many times a second,
        // Delta time makes sure that it's applying over time, so in one second it␣
↪will move scroll_speed pixels
        auto new_pos_x = pos.x() - constants.scroll_speed * timer::time_step::get_
↪fixed_delta_time();
```

(continues on next page)

(continued from previous page)

```
        // Set the new position value
        registry.replace<transform::position_2d>(pipe.body, new_pos_x, pos.y());

        // Set cap position too
        auto cap_pos = registry.get<transform::position_2d>(pipe.cap);
        registry.replace<transform::position_2d>(pipe.cap, new_pos_x, cap_pos.y());

        // Return the info about if this pipe is out of the screen
        return new_pos_x;
    }
};

// Name this system
REFL_AUTO (type(column_logic));

// Player Logic System
class player_logic final : public ecs::logic_update_system<player_logic> {
public:
    player_logic(entt::registry &registry, entt::entity player) noexcept :␣
↪system(registry), player_(player) {
        disable();
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the player
        auto pos = registry.get<transform::position_2d>(player_);

        // Add gravity to movement speed, multiply with delta time to apply it over␣
↪time
        movement_speed_.set_y(movement_speed_.y() + constants.gravity * timer::time_
↪step::get_fixed_delta_time());

        // Check if jump key is tapped
        bool jump_key_tapped = input::virtual_input::is_tapped("jump");

        // If jump is tapped, jump by adding jump force to the movement speed Y
        if (jump_key_tapped) movement_speed_.set_y(-constants.jump_force);

        // Add movement speed to position to make the character move, but apply over␣
↪time with delta time
        pos += movement_speed_ * timer::time_step::get_fixed_delta_time();

        // Do not let player to go out of the screen to top
        if (pos.y() <= 0.f) {
            pos.set_y(0.f);
            movement_speed_.set_y(0.f);
        }

        // Set the new position value
        registry.replace<transform::position_2d>(player_, pos);
```

(continues on next page)

(continued from previous page)

```
        // ROTATION
        // Retrieve props of the player
        auto &props = registry.get<transform::properties>(player_);

        // Increase the rotation a little by applying delta time
        float new_rotation = props.rotation + constants.rotate_speed * timer::time_
↪step::get_fixed_delta_time();

        // If jump button is tapped, reset rotation,
        // If rotation is higher than the max angle, set it to max angle
        if (jump_key_tapped)
            new_rotation = 0.f;
        else if (props.rotation > constants.max_angle)
            new_rotation = constants.max_angle;

        // Set the properties
        registry.replace<transform::properties>(player_, transform::properties{.
↪rotation = new_rotation});
    }

private:
    entt::entity player_;
    math::vec2f movement_speed_{0.f, 0.f};
};

// Name this system
REFL_AUTO (type(player_logic));

// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry, ecs::system_manager &system_manager)␣
↪noexcept : base_scene(registry), system_manager_(system_manager) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();

        // Create everything
        create_background(registry);
        init_dynamic_objects(registry);
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
        // Check if player requested to start the game
        check_start_game_request();
    }

    // Check if start game is requested at the pause state
    void check_start_game_request() {
        // If game is not started yet and jump key is tapped
```

(continues on next page)

(continued from previous page)

```
        if (!started_playing_ && input::virtual_input::is_tapped("jump")) {
            // Game starts, player started playing
            started_playing_ = true;
            resume_physics();
        }
    }

    // Initialize dynamic objects, this function is called at start and resets
    void init_dynamic_objects(entt::registry &registry) {
        create_columns(registry);

        // Create player
        auto player = create_player(registry);

        // Create logic systems
        create_logic_systems(player);

        // Reset state variables
        reset_state_variables();
    }

    // Create logic systems
    void create_logic_systems(entt::entity player) {
        system_manager_.create_system_rt<column_logic>();
        system_manager_.create_system_rt<player_logic>(player);
    }

    // Reset state values
    void reset_state_variables() {
        started_playing_ = false;
    }

    // Pause physics
    void pause_physics() {
        system_manager_.disable_systems<column_logic, player_logic>();
    }

    // Resume physics
    void resume_physics() {
        system_manager_.enable_systems<column_logic, player_logic>();
    }

    // System manager reference
    ecs::system_manager &system_manager_;

    // States
    bool started_playing_{false};
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
```

(continues on next page)

(continued from previous page)

```
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Create virtual input system
        system_manager_.create_system<ecs::virtual_input_system>();

        // Define the buttons for the jump action
        input::virtual_input::create("jump",
                                     {input::key::space, input::key::w,␣
↪input::key::up},
                                     {input::mouse_button::left, input::mouse_
↪button::right});

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_,␣
↪system_manager_), true);
    }
};

int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

Step 7: Collision between player and columns, death and reset game

Game ends when Flappy bird flies into the columns and dies. To set this up, we'll start with adding the collision system header `<antara/gaming/collisions/basic.collision.system.hpp>`.

Then make another logic system, `collision_logic`. Constructor gets the player entity and a reference of `player_dead` variable, so we can report back the collision result.

We store both in the class like this:

```
entt::entity player_;
bool &player_died_;
```

Then create the class and constructor:

```
// Collision Logic System
class collision_logic final : public ecs::logic_update_system<collision_logic> {
public:
    collision_logic(entt::registry &registry, entt::entity player_, bool &player_died_
↪) noexcept : system(registry),
                                                                                ␣
↪          player_(player_),
                                                                                ␣
↪          player_died_(player_died_) {}
```

(continues on next page)

(continued from previous page)

Now add a function to check collision between player and the pipes, `check_player_pipe_collision`.

Remember that we put columns to `layer<3>`. We can now retrieve them all by using `view` function, `registry.view<graphics::layer<3>>()`.

Then use `collisions::basic_collision_system::query_rect` function with `player_` and `entity` which is the pipe. If a collision is detected, we mark `player_died_` as `true`.

```
// Loop all columns to check collisions between player and the pipes
void check_player_pipe_collision(entt::registry &registry) {
    for (auto entity : registry.view<graphics::layer<3>>()) {
        // Check collision between player and a collidable object
        if (collisions::basic_collision_system::query_rect(registry, player_,␣
↪entity)) {
            // Mark player died as true
            player_died_ = true;
        }
    }
}
```

We'll call this function in the `update` function which is called every tick. But if `player_died_` is `true`, then no need to check for collision, we simply stop the function.

```
// Update, this will be called every tick
void update() noexcept final {
    auto &registry = entity_registry_;

    // Do not check anything if player is already dead
    if (player_died_) return;

    // Check collision
    check_player_pipe_collision(registry);
}
```

As we did earlier, we name this system, out of the class.

```
// Name this system
REFL_AUTO (type(collision_logic));
```

The completed class looks like this:

```
// Collision Logic System
class collision_logic final : public ecs::logic_update_system<collision_logic> {
public:
    collision_logic(entt::registry &registry, entt::entity player_, bool &player_died_
↪) noexcept : system(registry),
                                                                                ␣
↪          player_(player_),
                                                                                ␣
↪          player_died_(player_died_) {}
    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Do not check anything if player is already dead
```

(continues on next page)

(continued from previous page)

```
        if (player_died_) return;

        // Check collision
        check_player_pipe_collision(registry);
    }

private:
    // Loop all columns to check collisions between player and the pipes
    void check_player_pipe_collision(entt::registry &registry) {
        for (auto entity : registry.view<graphics::layer<3>>()) {
            // Check collision between player and a collidable object
            if (collisions::basic_collision_system::query_rect(registry, player_,␣
↪entity)) {
                // Mark player died as true
                player_died_ = true;
            }
        }
    }

    entt::entity player_;
    bool &player_died_;
};

// Name this system
REFL_AUTO (type(collision_logic));
```

Now let's use this class in `game_scene`:

```
// Create logic systems
void create_logic_systems(entt::entity player) {
    system_manager_.create_system_rt<column_logic>();
    system_manager_.create_system_rt<player_logic>(player);
    system_manager_.create_system_rt<collision_logic>(player, player_died_);
}
```

Then some more state variables for player death, game over, and reset query:

```
// States
bool started_playing_{false};
bool player_died_{false};
bool game_over_{false};
bool need_reset_{false};
```

Finally, add the needed values for game restart to `reset_state_variables`:

```
// Reset state values
void reset_state_variables() {
    started_playing_ = false;
    player_died_ = false;
    game_over_ = false;
}
```

Since `player_died_` will be filled by `collision_logic`, we can read it in this class. When it's `true`, we will mark `game_over_` as `true` and pause physics because we want the game to stop when player dies. We'll also mark `player_died_` to `false` so these won't be triggered again.

```
// Check if player died
void check_death() {
    // If player died, game over, and pause physics
    if (player_died_) {
        player_died_ = false;
        game_over_ = true;
        pause_physics();
    }
}
```

Another function will check for a jump button press after the game is over. When jump button is pressed, game will restart.

```
// Check if reset is requested at game over state
void check_reset_request() {
    // If game is over, and jump key is pressed, reset game
    if (game_over_ && input::virtual_input::is_tapped("jump")) reset_game();
}
```

Let's call these two in the `update` function:

```
// Update the game every tick
void update() noexcept final {
    // Check if player requested to start the game
    check_start_game_request();

    // Check if player died
    check_death();

    // Check if player requested reset after death
    check_reset_request();
}
```

As you saw in `check_reset_request`, we use `reset_game` function, so let's define that:

```
// Reset game
void reset_game() {
    // Destroy all dynamic objects
    destroy_dynamic_objects();

    // Queue reset to reinitialize
    this->need_reset_ = true;
}
```

In `reset_game` we want to destroy dynamic objects. To do that, we retrieve all the dynamic entities with `dynamic` tag that we set before, then destroy them all using the registry.

For logic system deletions, we need to mark items for deletion, with the function below:

```
// Destroy dynamic objects
void destroy_dynamic_objects() {
    // Retrieve the collection of entities from the game scene
    auto view = entity_registry_.view<entt::tag<"dynamic"_hs>>();

    // Iterate the collection and destroy each entities
    entity_registry_.destroy(view.begin(), view.end());
```

(continues on next page)

(continued from previous page)

```
    // Delete systems
    system_manager_.mark_systems<player_logic, collision_logic>();
}
```

Those systems get deleted after the whole update tick is completed, so we don't want to reinitialize them in `reset_game`. Instead, queue the reset by setting `need_reset_` true, and do reinitialization in `post_update` like this:

```
// Post update
void post_update() noexcept final {
    // If reset is requested
    if (need_reset_) {
        // Reinitialize all these
        init_dynamic_objects(entity_registry_);
        need_reset_ = false;
    }
}
```

That's it! Flappy bird now collides with pipes, dies, and enters the "game over" state. Afterwards, by pressing jump button, all the dynamic entities and logic systems are destroyed, then reinitialized.

Step 7 is complete, here is the full code.

```
#include <random>
#include <antara/gaming/ecs/virtual.input.system.hpp>
#include <antara/gaming/collisions/basic.collision.system.hpp>
#include <antara/gaming/graphics/component.layer.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/graphics/component.sprite.hpp>
#include <antara/gaming/input/virtual.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Constants
struct flappy_bird_constants {
    // Player
    const std::string player_image_name{"player.png"};
    const float player_pos_x{400.0f};
    const float gravity{2000.f};
    const float jump_force{650.f};
    const float rotate_speed{100.f};
    const float max_angle{60.f};

    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
```

(continues on next page)

(continued from previous page)

```
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};
    const float scroll_speed{200.f};

    // Background
    const float ground_thickness{100.0f};
    const float grass_thickness{20.0f};
    const graphics::color background_color{82, 189, 199};
    const graphics::color ground_color{220, 209, 143};
    const graphics::color grass_color{132, 227, 90};
    const graphics::outline_color grass_outline_color{2.0f, graphics::color{76, 47,␣
↪61}};
};

// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}

// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
    entt::entity cap{entt::null};

    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};

// Column is made of two pipes
struct column {
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
        top_pipe.destroy(registry);
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
};

// Logic functions
```

(continues on next page)

(continued from previous page)

```
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }

    // Returns a random gap start position Y
    float get_random_gap_start_pos(const entt::registry &registry) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        float top_limit = canvas_height * constants.column_min;
        float bottom_limit = canvas_height * constants.column_max - constants.gap_
↪height;

        return random_float(top_limit, bottom_limit);
    }
}

// Factory functions
namespace {
    // Factory for pipes, requires to know if it's a top one, position x of the␣
↪column, and the gap starting position Y
    pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_
↪start_pos_y) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        // PIPE BODY
        // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the␣
↪canvas
        transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

        // Size X is the column thickness,
        // Size Y is the important part.
        // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
        //  So half size should be gap_start_pos_y since center of the rectangle is␣
↪at 0.
        // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y +␣
↪gap_height
        //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
        // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
        math::vec2f body_size{constants.column_thickness,
                              is_top ?
                              gap_start_pos_y * 2.0f :
                              (canvas_height - (gap_start_pos_y + constants.gap_
↪height)) * 2.0f};
```

(continues on next page)

(continued from previous page)

```
        auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                                  constants.pipe_outline_color);

        // PIPE CAP
        // Let's prepare the pipe cap
        // Size of the cap is defined in constants
        math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_
↪width, constants.pipe_cap_height};

        // Position, X is same as the body. Bottom of the cap is aligned with bottom␣
↪of the body,
        // or start of the gap, we will use start of the gap here, minus half of the␣
↪cap height
        transform::position_2d cap_pos{body_pos.x(),
                                       is_top ?
                                       gap_start_pos_y - constants.pipe_cap_height *␣
↪0.5f :
                                       gap_start_pos_y + constants.gap_height +␣
↪constants.pipe_cap_height * 0.5f
        };

        // Construct the cap
        auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_
↪color, cap_pos,
                                                 constants.pipe_outline_color);

        // Set layers, cap should be in front of body
        registry.assign<graphics::layer<4>>(cap);
        registry.assign<graphics::layer<3>>(body);
        tag_game_scene(registry, cap, true);
        tag_game_scene(registry, body, true);

        // Construct a pipe with body and cap and return it
        return {body, cap};
    }

    // Factory to create single column
    void create_column(entt::registry &registry, float pos_x) noexcept {
        // Create a fresh entity for a new column
        auto entity_column = registry.create();

        // Get a random gap start position Y, between pipes
        float gap_start_pos_y = get_random_gap_start_pos(registry);

        // Create pipes, is_top variable is false for bottom one
        auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
        auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

        // Make a column from these two pipes and mark it as "column"
        registry.assign<column>(entity_column, top_pipe, bottom_pipe);
        registry.assign<entt::tag<"column"_hs>>(entity_column);
        tag_game_scene(registry, entity_column, true);
    }

    // Factory for creating a Flappy Bird columns
    void create_columns(entt::registry &registry) noexcept {
```

(continues on next page)

(continued from previous page)

```
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Spawn columns out of the screen, out of the canvas
        const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

        // Create the columns
        for (std::size_t i = 0; i < constants.column_count; ++i) {
            // Horizontal position (X) increases for every column, keeping the␣
↪distance
            float pos_x = column_pos_offset + i * constants.column_distance;

            create_column(registry, pos_x);
        }
    }

    // Factory for creating a Flappy Bird background
    void create_background(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Create Sky
        {
            // Sky is whole canvas so position is middle of it
            transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

            // And the size is full canvas
            math::vec2f size{canvas_width, canvas_height};

            auto sky = geometry::blueprint_rectangle(registry, size, constants.
↪background_color, pos);
            registry.assign<graphics::layer<1>>(sky);
            tag_game_scene(registry, sky);
        }

        // Create Grass
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at top of the ground, so it's canvas height minus␣
↪ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness};

            // Size X is full canvas but the height is defined in constants
            // We also make it a bit longer by adding the thickness of the outline to␣
↪hide the outline at sides
            math::vec2f size{canvas_width + constants.grass_outline_color.thickness *␣
↪2.0f, constants.grass_thickness};

            auto grass = geometry::blueprint_rectangle(registry, size, constants.
↪grass_color, pos,
                                                       constants.grass_outline_color);
            registry.assign<graphics::layer<3>>(grass);
            tag_game_scene(registry, grass);
```

(continues on next page)

(continued from previous page)

```
        }

        // Create Ground
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at bottom of the screen so it's full canvas_height␣
↪minus half of the ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness * 0.5f};

            // Size X is full canvas but the height is defined in constants
            math::vec2f size{canvas_width, constants.ground_thickness};

            auto ground = geometry::blueprint_rectangle(registry, size, constants.
↪ground_color, pos);
            registry.assign<graphics::layer<3>>(ground);
            tag_game_scene(registry, ground);
        }
    }

    // Factory for creating the player
    entt::entity create_player(entt::registry &registry) {
        // Retrieve constants
        const auto[_, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.
↪size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        auto entity = graphics::blueprint_sprite(registry,
                                                 graphics::sprite{constants.player_
↪image_name.c_str()},
                                                 transform::position_2d{constants.
↪player_pos_x, canvas_height * 0.5f});
        registry.assign<antara::gaming::graphics::layer<5>>(entity);
        registry.assign<entt::tag<"player"_hs>>(entity);
        tag_game_scene(registry, entity, true);

        return entity;
    }
}

// Column Logic System
class column_logic final : public ecs::logic_update_system<column_logic> {
public:
    explicit column_logic(entt::registry &registry) noexcept : system(registry) {
        disable();
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Loop all columns
        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);
```

(continues on next page)

(continued from previous page)

```
            // Move pipes, and retrieve column position x
            float column_pos_x = move_pipe(registry, col.top_pipe);
            move_pipe(registry, col.bottom_pipe);

            // If column is out of the screen
            if (column_pos_x < -constants.column_distance) {
                // Remove this column
                col.destroy(registry, entity);

                // Create a new column at far end
                create_column(registry, furthest_pipe_position(registry) + constants.
↪column_distance);
            }
        }
    }

private:
    // Find the furthest pipe's position X
    float furthest_pipe_position(entt::registry &registry) {
        float furthest = 0.f;

        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);
            float x = entity_registry_.get<transform::position_2d>(col.top_pipe.body).
↪x();
            if (x > furthest) furthest = x;
        }

        return furthest;
    }

    // Move the pipe and return the x position
    float move_pipe(entt::registry &registry, pipe &pipe) {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the pipe
        auto pos = registry.get<transform::position_2d>(pipe.body);

        // Shift pos X to left by scroll_speed but multiplying with dt because we do␣
↪this so many times a second,
        // Delta time makes sure that it's applying over time, so in one second it␣
↪will move scroll_speed pixels
        auto new_pos_x = pos.x() - constants.scroll_speed * timer::time_step::get_
↪fixed_delta_time();

        // Set the new position value
        registry.replace<transform::position_2d>(pipe.body, new_pos_x, pos.y());

        // Set cap position too
        auto cap_pos = registry.get<transform::position_2d>(pipe.cap);
        registry.replace<transform::position_2d>(pipe.cap, new_pos_x, cap_pos.y());

        // Return the info about if this pipe is out of the screen
        return new_pos_x;
    }
```

(continues on next page)

(continued from previous page)

```
};

// Name this system
REFL_AUTO (type(column_logic));

// Player Logic System
class player_logic final : public ecs::logic_update_system<player_logic> {
public:
    player_logic(entt::registry &registry, entt::entity player) noexcept :␣
↪system(registry), player_(player) {
        disable();
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the player
        auto pos = registry.get<transform::position_2d>(player_);

        // Add gravity to movement speed, multiply with delta time to apply it over␣
↪time
        movement_speed_.set_y(movement_speed_.y() + constants.gravity * timer::time_
↪step::get_fixed_delta_time());

        // Check if jump key is tapped
        bool jump_key_tapped = input::virtual_input::is_tapped("jump");

        // If jump is tapped, jump by adding jump force to the movement speed Y
        if (jump_key_tapped) movement_speed_.set_y(-constants.jump_force);

        // Add movement speed to position to make the character move, but apply over␣
↪time with delta time
        pos += movement_speed_ * timer::time_step::get_fixed_delta_time();

        // Do not let player to go out of the screen to top
        if (pos.y() <= 0.f) {
            pos.set_y(0.f);
            movement_speed_.set_y(0.f);
        }

        // Set the new position value
        registry.replace<transform::position_2d>(player_, pos);

        // ROTATION
        // Retrieve props of the player
        auto &props = registry.get<transform::properties>(player_);

        // Increase the rotation a little by applying delta time
        float new_rotation = props.rotation + constants.rotate_speed * timer::time_
↪step::get_fixed_delta_time();

        // If jump button is tapped, reset rotation,
        // If rotation is higher than the max angle, set it to max angle
```

(continues on next page)

(continued from previous page)

```
        if (jump_key_tapped)
            new_rotation = 0.f;
        else if (props.rotation > constants.max_angle)
            new_rotation = constants.max_angle;

        // Set the properties
        registry.replace<transform::properties>(player_, transform::properties{.
↪rotation = new_rotation});
    }

private:
    entt::entity player_;
    math::vec2f movement_speed_{0.f, 0.f};
};

// Name this system
REFL_AUTO (type(player_logic));

// Collision Logic System
class collision_logic final : public ecs::logic_update_system<collision_logic> {
public:
    collision_logic(entt::registry &registry, entt::entity player, bool &player_died)␣
↪noexcept : system(registry),
                                                                                ␣
↪              player_(player),
                                                                                ␣
↪              player_died_(player_died) {}
    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Do not check anything if player is already dead
        if (player_died_) return;

        // Check collision
        check_player_pipe_collision(registry);
    }

private:
    // Loop all columns to check collisions between player and the pipes
    void check_player_pipe_collision(entt::registry &registry) {
        for (auto entity : registry.view<graphics::layer<3>>()) {
            // Check collision between player and a collidable object
            if (collisions::basic_collision_system::query_rect(registry, player_,␣
↪entity)) {
                // Mark player died as true
                player_died_ = true;
            }
        }
    }

    entt::entity player_;
    bool &player_died_;
};

// Name this system
REFL_AUTO (type(collision_logic));
```

(continues on next page)

(continued from previous page)

```
// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry, ecs::system_manager &system_manager)␣
↪noexcept : base_scene(registry),

↪     system_manager_(system_manager) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();

        // Create everything
        create_background(registry);
        init_dynamic_objects(registry);
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
        // Check if player requested to start the game
        check_start_game_request();

        // Check if player died
        check_death();

        // Check if player requested reset after death
        check_reset_request();
    }

    // Check if start game is requested at the pause state
    void check_start_game_request() {
        // If game is not started yet and jump key is tapped
        if (!started_playing_ && input::virtual_input::is_tapped("jump")) {
            // Game starts, player started playing
            started_playing_ = true;
            resume_physics();
        }
    }

    // Check if player died
    void check_death() {
        // If player died, game over, and pause physics
        if (player_died_) {
            player_died_ = false;
            game_over_ = true;
            pause_physics();
        }
    }

    // Check if reset is requested at game over state
    void check_reset_request() {
        // If game is over, and jump key is pressed, reset game
```

(continues on next page)

(continued from previous page)

```
    if (game_over_ && input::virtual_input::is_tapped("jump")) reset_game();
}

// Initialize dynamic objects, this function is called at start and resets
void init_dynamic_objects(entt::registry &registry) {
    create_columns(registry);

    // Create player
    auto player = create_player(registry);

    // Create logic systems
    create_logic_systems(player);

    // Reset state variables
    reset_state_variables();
}

// Create logic systems
void create_logic_systems(entt::entity player) {
    system_manager_.create_system_rt<column_logic>();
    system_manager_.create_system_rt<player_logic>(player);
    system_manager_.create_system_rt<collision_logic>(player, player_died_);
}

// Reset state values
void reset_state_variables() {
    started_playing_ = false;
    player_died_ = false;
    game_over_ = false;
}

// Pause physics
void pause_physics() {
    system_manager_.disable_systems<column_logic, player_logic>();
}

// Resume physics
void resume_physics() {
    system_manager_.enable_systems<column_logic, player_logic>();
}

// Destroy dynamic objects
void destroy_dynamic_objects() {
    // Retrieve the collection of entities from the game scene
    auto view = entity_registry_.view<entt::tag<"dynamic"_hs>>();

    // Iterate the collection and destroy each entities
    entity_registry_.destroy(view.begin(), view.end());

    // Delete systems
    system_manager_.mark_systems<player_logic, collision_logic>();
}

// Reset game
void reset_game() {
    // Destroy all dynamic objects
    destroy_dynamic_objects();
```

(continues on next page)

(continued from previous page)

```
        // Queue reset to reinitialize
        this->need_reset_ = true;
    }

    // Post update
    void post_update() noexcept final {
        // If reset is requested
        if (need_reset_) {
            // Reinitialize all these
            init_dynamic_objects(entity_registry_);
            need_reset_ = false;
        }
    }

    // System manager reference
    ecs::system_manager &system_manager_;

    // States
    bool started_playing_{false};
    bool player_died_{false};
    bool game_over_{false};
    bool need_reset_{false};
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Create virtual input system
        system_manager_.create_system<ecs::virtual_input_system>();

        // Define the buttons for the jump action
        input::virtual_input::create("jump",
                                     {input::key::space, input::key::w,␣
↪input::key::up},
                                     {input::mouse_button::left, input::mouse_
↪button::right});

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_,␣
↪system_manager_), true);
    }
};
```

(continues on next page)

(continued from previous page)

```
int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

Step 8: Score and UI

Without scores there is no measure of achievement, and no motivation to play again to see if your skills have improved (or are better than your friends!). So we will count scores and display it on the screen.

Only one constant is needed here, `font_size`.

```
struct flappy_bird_constants {
    // UI
    const unsigned long long font_size{32ull};
```

We will get one score from each column player passes, so we need to mark the column as `scored`. Let's put a variable into the `struct column`.

```
// Is score taken from this column
bool scored{false};
```

Then we define a score struct, it will have the current score, max score and UI text.

```
// Score struct, has current value, max record, and the UI text
struct score {
    int value;
    int max_score;
    entt::entity text;
};
```

Now make a function which constructs the UI text, and put it under Logic functions `namespace`:

```
// Create the UI string
std::string score_ui_text(int score = 0, int best_score = 0) {
    return "Score: "s + std::to_string(score) +
           "\nBest: "s + std::to_string(best_score) +
           "\n\nW / UP / Space / Mouse to FLAP"s;
}
```

Next we can make the `create_score` function which will make an entity.

Retrieve constants and canvas size:

```
// Factory to create score entity
entt::entity create_score(entt::registry &registry) {
    // Retrieve constants
    const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
    const auto constants = registry.ctx<flappy_bird_constants>();
```

Then, create `text_entity` using `graphics::blueprint_text` function, feed the text `score_ui_text` and `font_size` from `constants`.

```
// Create text
auto text_entity = graphics::blueprint_text(registry, graphics::text{score_ui_text(),␣
↪constants.font_size},
    transform::position_2d{canvas_width * 0.03f, canvas_height * 0.03f},␣
↪graphics::white);
```

Set it ot `layer<9>` because we want the text to be in front of everything, and tag it as `game_scene`.

```
registry.assign<graphics::layer<9>>(text_entity);
tag_game_scene(registry, text_entity);
```

Now we create a fresh entity, assign `struct score` to it with 0 score and max record values, and `text_entity` we just created. Tag it as `high_score` and `game_scene`, then return it.

```
// Create a fresh entity
auto entity = registry.create();

// Create score
registry.assign<score>(entity, 0, 0, text_entity);
registry.assign<entt::tag<"high_score"_hs>>(entity);
tag_game_scene(registry, entity);

return entity;
```

Add a member for it in `game_scene`:

```
entt::entity score_entity_;
```

Then create it inside `game_scene` constructor using the `create_score` function:

```
game_scene(entt::registry &registry, ecs::system_manager &system_manager) noexcept :␣
↪base_scene(registry),
                                                                                ␣
↪system_manager_(system_manager) {
    // Set the constants that will be used in the program
    registry.set<flappy_bird_constants>();

    // Create everything
    score_entity_ = create_score(registry);
    create_background(registry);
    init_dynamic_objects(registry);
}
```

Now let's make the function which will update the score. This function will be able to do two things: increment score by one, and reset the score when game is being reset.

We simply have a parameter `reset` to know about the reset situation.

First we retrieve the `struct score` from the entity, then if reset is requested, we simply set the `value` to zero.

If reset is not requested, then it will increment `value` by one, check if it's higher than the `max_score`, and update `max_score` if `value` it is.

```
void update_score(entt::registry &registry, entt::entity entity, bool reset = false) {
    score &sc = registry.get<score>(entity);
```

(continues on next page)

(continued from previous page)

```
    // If reset is asked, set score to 0
    if (reset) sc.value = 0;
        // Else, increase the score,
        // Compare it with the max score, and update max score if it's greater
    else if (++sc.value > sc.max_score) sc.max_score = sc.value;
```

Next, update the `struct score` inside the score entity.

```
// Update the score entity
registry.replace<score>(entity, sc);
```

Then then update the contents of `graphics::text` with the `score_ui_text` using the new values.

```
// Update the UI text entity with the current values
auto &text = registry.get<graphics::text>(sc.text);
text.contents = score_ui_text(sc.value, sc.max_score);
registry.replace<graphics::text>(sc.text, text);
```

The completed function looks like this:

```
// Update score
void update_score(entt::registry &registry, entt::entity entity, bool reset = false) {
    score &sc = registry.get<score>(entity);

    // If reset is asked, set score to 0
    if (reset) sc.value = 0;
        // Else, increase the score,
        // Compare it with the max score, and update max score if it's greater
    else if (++sc.value > sc.max_score) sc.max_score = sc.value;

    // Update the score entity
    registry.replace<score>(entity, sc);

    // Update the UI text entity with the current values
    auto &text = registry.get<graphics::text>(sc.text);
    text.contents = score_ui_text(sc.value, sc.max_score);
    registry.replace<graphics::text>(sc.text, text);
}
```

We have the function to update the score now, and it needs to be called when player passes a column. This function needs the score entity so we will pass it to `column_logic` with the constructor.

First, have a class member for entity.

```
entt::entity score_entity_;
```

Then fill it with the constructor.

```
// Column Logic System
class column_logic final : public ecs::logic_update_system<column_logic> {
public:
    explicit column_logic(entt::registry &registry, entt::entity score) noexcept :␣
↪system(registry),
                                                                                score_
↪entity_(score) {
        disable();
```

(continues on next page)

(continued from previous page)

```
    }
```

We need to update the creation line too, feeding the score entity.

```
void create_logic_systems(entt::entity player) {
    system_manager_.create_system_rt<column_logic>(score_entity_);
```

Now we go back to the `update` function of this class, and inside the for loop which loops all columns, we add the check for score.

At first, column should be new, with `score` field being false. Once the column position is to the left side of the player position (after Flappy bird passes), use a simple < comparison of the column and Flappy bird's position on the X axis.

Inside, we call the `update_score` function, and mark the column `scored` as `true`.

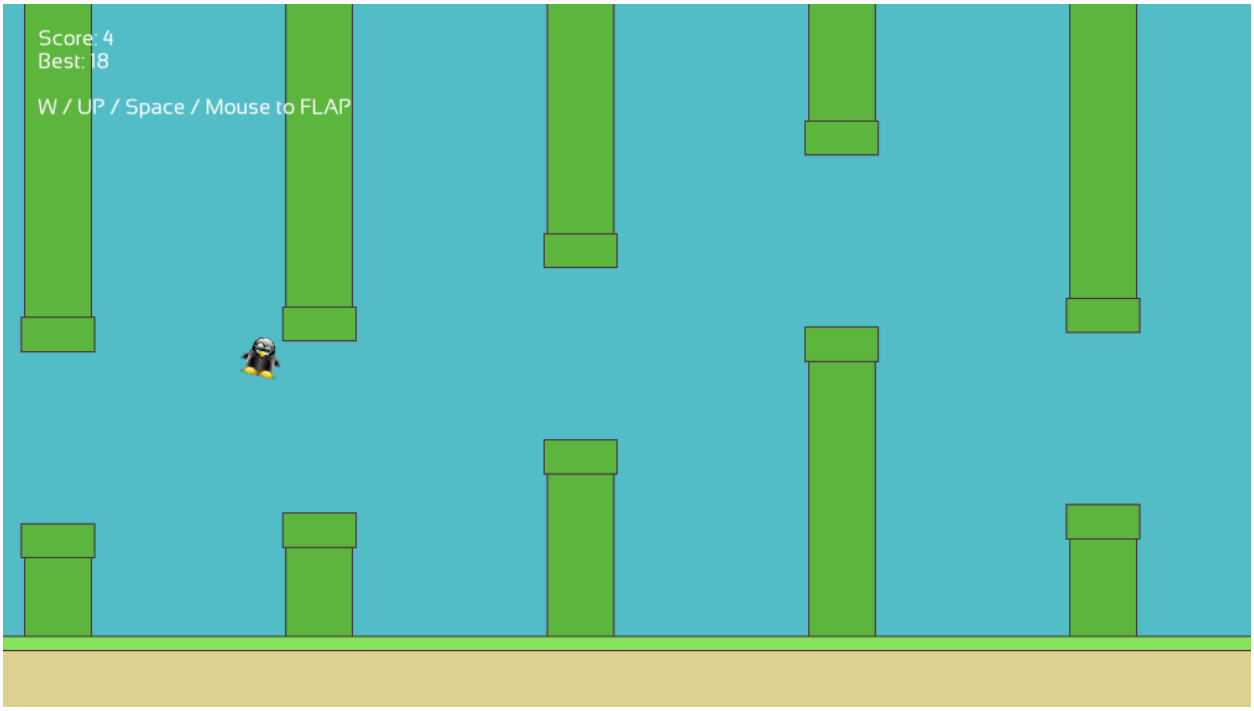
```
// If this column is not scored, and player passed this column
if (!col.scored && column_pos_x < constants.player_pos_x) {
    // Increase the score
    update_score(registry, score_entity_);

    // Set column as scored
    col.scored = true;
}
```

Great, score is being counted now.

Next thing we want is to reset this score value when game is over and `reset_game` is called. We use `update_score` function to do this, but this time we set the last parameter, `reset` as `true`.

```
// Reset game
void reset_game() {
    // Destroy all dynamic objects
    destroy_dynamic_objects();

    // Queue reset to reinitialize
    this->need_reset_ = true;

    // Reset current score, but keep the max score
    update_score(entity_registry_, score_entity_, true);
}
```

That's it, now if you run the game, you'll see the UI which shows current score, max score and button instructions, and as you play, you'll see score and max score increasing, and see the score return to zero when you die and reset the game.



Step 8 is complete, here is the full code.

```
#include <random>
#include <antara/gaming/ecs/virtual.input.system.hpp>
#include <antara/gaming/collisions/basic.collision.system.hpp>
#include <antara/gaming/graphics/component.layer.hpp>
#include <antara/gaming/graphics/component.canvas.hpp>
#include <antara/gaming/math/vector.hpp>
#include <antara/gaming/scenes/scene.manager.hpp>
#include <antara/gaming/sfml/graphic.system.hpp>
#include <antara/gaming/sfml/input.system.hpp>
#include <antara/gaming/sfml/resources.manager.hpp>
#include <antara/gaming/world/world.app.hpp>
#include <antara/gaming/graphics/component.sprite.hpp>
#include <antara/gaming/input/virtual.hpp>

// For convenience
using namespace antara::gaming;
using namespace std::string_literals;

// Constants
struct flappy_bird_constants {
    // Player
    const std::string player_image_name{"player.png"};
    const float player_pos_x{400.0f};
    const float gravity{2000.f};
    const float jump_force{650.f};
    const float rotate_speed{100.f};
    const float max_angle{60.f};

    // Pipes
    const float gap_height{265.f};
    const float column_start_distance{700.f};
```

(continues on next page)

(continued from previous page)

```
    const float column_min{0.2f};
    const float column_max{0.8f};
    const float column_thickness{100.f};
    const float column_distance{400.f};
    const std::size_t column_count{6};
    const float pipe_cap_extra_width{10.f};
    const float pipe_cap_height{50.f};
    const graphics::color pipe_color{92, 181, 61};
    const graphics::outline_color pipe_outline_color{2.0f, graphics::color{76, 47, 61}
↪};
    const float scroll_speed{200.f};

    // Background
    const float ground_thickness{100.0f};
    const float grass_thickness{20.0f};
    const graphics::color background_color{82, 189, 199};
    const graphics::color ground_color{220, 209, 143};
    const graphics::color grass_color{132, 227, 90};
    const graphics::outline_color grass_outline_color{2.0f, graphics::color{76, 47,␣
↪61}};
};

// Random number generator
namespace {
    std::random_device rd;  // Will be used to obtain a seed for the random number␣
↪engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    float random_float(float lower, float higher) {
        std::uniform_real_distribution<float> dist(lower, higher);
        return dist(gen);
    }
}

// A Flappy Bird column which has two pipes
struct pipe {
    entt::entity body{entt::null};
    entt::entity cap{entt::null};

    // Destroy pipe
    void destroy(entt::registry &registry) {
        registry.destroy(body);
        registry.destroy(cap);
    }
};

// Column is made of two pipes
struct column {
    // Entities representing the Flappy Bird pipes
    pipe top_pipe{entt::null};
    pipe bottom_pipe{entt::null};

    // Destroy pipes and this column
    void destroy(entt::registry &registry, entt::entity entity) {
        top_pipe.destroy(registry);
        bottom_pipe.destroy(registry);
        registry.destroy(entity);
    }
```

(continues on next page)

(continued from previous page)

```
};

// Logic functions
namespace {
    void tag_game_scene(entt::registry &registry, entt::entity entity, bool dynamic =␣
↪false) {
        // Tag game scene
        registry.assign<entt::tag<"game_scene"_hs>>(entity);

        // Tag dynamic
        if(dynamic) registry.assign<entt::tag<"dynamic"_hs>>(entity);
    }

    // Returns a random gap start position Y
    float get_random_gap_start_pos(const entt::registry &registry) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        float top_limit = canvas_height * constants.column_min;
        float bottom_limit = canvas_height * constants.column_max - constants.gap_
↪height;

        return random_float(top_limit, bottom_limit);
    }
}

// Factory functions
namespace {
    // Factory for pipes, requires to know if it's a top one, position x of the␣
↪column, and the gap starting position Y
    pipe create_pipe(entt::registry &registry, bool is_top, float pos_x, float gap_
↪start_pos_y) {
        // Retrieve constants
        const auto canvas_height = registry.ctx<graphics::canvas_2d>().canvas.size.
↪y();
        const auto constants = registry.ctx<flappy_bird_constants>();

        // PIPE BODY
        // Top pipe is at Y: 0 and bottom pipe is at canvas_height, bottom of the␣
↪canvas
        transform::position_2d body_pos{pos_x, is_top ? 0.f : canvas_height};

        // Size X is the column thickness,
        // Size Y is the important part.
        // If it's a top pipe, gap_start_pos_y should be bottom of the rectangle
        //  So half size should be gap_start_pos_y since center of the rectangle is␣
↪at 0.
        // If it's the bottom pipe, top of the rectangle will be at gap_start_pos_y +␣
↪gap_height
        //  So half size should be canvas_height - (gap_start_pos_y + gap_height)
        // Since these are half-sizes, and the position is at the screen border, we␣
↪multiply these sizes by two
        math::vec2f body_size{constants.column_thickness,
                              is_top ?
                              gap_start_pos_y * 2.0f :
```

(continues on next page)

(continued from previous page)

```
                                (canvas_height - (gap_start_pos_y + constants.gap_
↪height)) * 2.0f};

        auto body = geometry::blueprint_rectangle(registry, body_size, constants.pipe_
↪color, body_pos,
                                                  constants.pipe_outline_color);

        // PIPE CAP
        // Let's prepare the pipe cap
        // Size of the cap is defined in constants
        math::vec2f cap_size{constants.column_thickness + constants.pipe_cap_extra_
↪width, constants.pipe_cap_height};

        // Position, X is same as the body. Bottom of the cap is aligned with bottom␣
↪of the body,
        // or start of the gap, we will use start of the gap here, minus half of the␣
↪cap height
        transform::position_2d cap_pos{body_pos.x(),
                                       is_top ?
                                       gap_start_pos_y - constants.pipe_cap_height *␣
↪0.5f :
                                       gap_start_pos_y + constants.gap_height +␣
↪constants.pipe_cap_height * 0.5f
        };

        // Construct the cap
        auto cap = geometry::blueprint_rectangle(registry, cap_size, constants.pipe_
↪color, cap_pos,
                                                 constants.pipe_outline_color);

        // Set layers, cap should be in front of body
        registry.assign<graphics::layer<4>>(cap);
        registry.assign<graphics::layer<3>>(body);
        tag_game_scene(registry, cap, true);
        tag_game_scene(registry, body, true);

        // Construct a pipe with body and cap and return it
        return {body, cap};
    }

    // Factory to create single column
    void create_column(entt::registry &registry, float pos_x) noexcept {
        // Create a fresh entity for a new column
        auto entity_column = registry.create();

        // Get a random gap start position Y, between pipes
        float gap_start_pos_y = get_random_gap_start_pos(registry);

        // Create pipes, is_top variable is false for bottom one
        auto top_pipe = create_pipe(registry, true, pos_x, gap_start_pos_y);
        auto bottom_pipe = create_pipe(registry, false, pos_x, gap_start_pos_y);

        // Make a column from these two pipes and mark it as "column"
        registry.assign<column>(entity_column, top_pipe, bottom_pipe);
        registry.assign<entt::tag<"column"_hs>>(entity_column);
        tag_game_scene(registry, entity_column, true);
    }
```

(continues on next page)

(continued from previous page)

```
    // Factory for creating a Flappy Bird columns
    void create_columns(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Spawn columns out of the screen, out of the canvas
        const float column_pos_offset = constants.column_start_distance + constants.
↪column_thickness * 2.0f;

        // Create the columns
        for (std::size_t i = 0; i < constants.column_count; ++i) {
            // Horizontal position (X) increases for every column, keeping the␣
↪distance
            float pos_x = column_pos_offset + i * constants.column_distance;

            create_column(registry, pos_x);
        }
    }

    // Factory for creating a Flappy Bird background
    void create_background(entt::registry &registry) noexcept {
        // Retrieve constants
        const auto[canvas_width, canvas_height] = registry.ctx<graphics::canvas_2d>().
↪canvas.size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Create Sky
        {
            // Sky is whole canvas so position is middle of it
            transform::position_2d pos{canvas_width * 0.5f, canvas_height * 0.5f};

            // And the size is full canvas
            math::vec2f size{canvas_width, canvas_height};

            auto sky = geometry::blueprint_rectangle(registry, size, constants.
↪background_color, pos);
            registry.assign<graphics::layer<1>>(sky);
            tag_game_scene(registry, sky);
        }

        // Create Grass
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at top of the ground, so it's canvas height minus␣
↪ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness};

            // Size X is full canvas but the height is defined in constants
            // We also make it a bit longer by adding the thickness of the outline to␣
↪hide the outline at sides
            math::vec2f size{canvas_width + constants.grass_outline_color.thickness *␣
↪2.0f, constants.grass_thickness};

            auto grass = geometry::blueprint_rectangle(registry, size, constants.
↪grass_color, pos,
```

(continues on next page)

(continued from previous page)

```
                                                        constants.grass_outline_color);
            registry.assign<graphics::layer<3>>(grass);
            tag_game_scene(registry, grass);
        }

        // Create Ground
        {
            // Ground expands to whole canvas width so position is middle of it,
            // But position Y is at bottom of the screen so it's full canvas_height␣
↪minus half of the ground thickness
            transform::position_2d pos{canvas_width * 0.5f, canvas_height - constants.
↪ground_thickness * 0.5f};

            // Size X is full canvas but the height is defined in constants
            math::vec2f size{canvas_width, constants.ground_thickness};

            auto ground = geometry::blueprint_rectangle(registry, size, constants.
↪ground_color, pos);
            registry.assign<graphics::layer<3>>(ground);
            tag_game_scene(registry, ground);
        }
    }

    // Factory for creating the player
    entt::entity create_player(entt::registry &registry) {
        // Retrieve constants
        const auto[_, canvas_height] = registry.ctx<graphics::canvas_2d>().canvas.
↪size;
        const auto constants = registry.ctx<flappy_bird_constants>();

        auto entity = graphics::blueprint_sprite(registry,
                                                 graphics::sprite{constants.player_
↪image_name.c_str()},
                                                 transform::position_2d{constants.
↪player_pos_x, canvas_height * 0.5f});
        registry.assign<antara::gaming::graphics::layer<5>>(entity);
        registry.assign<entt::tag<"player"_hs>>(entity);
        tag_game_scene(registry, entity, true);

        return entity;
    }
}

// Column Logic System
class column_logic final : public ecs::logic_update_system<column_logic> {
public:
    explicit column_logic(entt::registry &registry) noexcept : system(registry) {
        disable();
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();
```

(continues on next page)

(continued from previous page)

```
        // Loop all columns
        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);

            // Move pipes, and retrieve column position x
            float column_pos_x = move_pipe(registry, col.top_pipe);
            move_pipe(registry, col.bottom_pipe);

            // If column is out of the screen
            if (column_pos_x < -constants.column_distance) {
                // Remove this column
                col.destroy(registry, entity);

                // Create a new column at far end
                create_column(registry, furthest_pipe_position(registry) + constants.
↪column_distance);
            }
        }
    }

private:
    // Find the furthest pipe's position X
    float furthest_pipe_position(entt::registry &registry) {
        float furthest = 0.f;

        for (auto entity : registry.view<column>()) {
            auto &col = registry.get<column>(entity);
            float x = entity_registry_.get<transform::position_2d>(col.top_pipe.body).
↪x();
            if (x > furthest) furthest = x;
        }

        return furthest;
    }

    // Move the pipe and return the x position
    float move_pipe(entt::registry &registry, pipe &pipe) {
        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the pipe
        auto pos = registry.get<transform::position_2d>(pipe.body);

        // Shift pos X to left by scroll_speed but multiplying with dt because we do␣
↪this so many times a second,
        // Delta time makes sure that it's applying over time, so in one second it␣
↪will move scroll_speed pixels
        auto new_pos_x = pos.x() - constants.scroll_speed * timer::time_step::get_
↪fixed_delta_time();

        // Set the new position value
        registry.replace<transform::position_2d>(pipe.body, new_pos_x, pos.y());

        // Set cap position too
        auto cap_pos = registry.get<transform::position_2d>(pipe.cap);
        registry.replace<transform::position_2d>(pipe.cap, new_pos_x, cap_pos.y());
```

(continues on next page)

(continued from previous page)

```
        // Return the info about if this pipe is out of the screen
        return new_pos_x;
    }
};

// Name this system
REFL_AUTO (type(column_logic));

// Player Logic System
class player_logic final : public ecs::logic_update_system<player_logic> {
public:
    player_logic(entt::registry &registry, entt::entity player) noexcept :␣
↪system(registry), player_(player) {
        disable();
    }

    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Retrieve constants
        const auto constants = registry.ctx<flappy_bird_constants>();

        // Get current position of the player
        auto pos = registry.get<transform::position_2d>(player_);

        // Add gravity to movement speed, multiply with delta time to apply it over␣
↪time
        movement_speed_.set_y(movement_speed_.y() + constants.gravity * timer::time_
↪step::get_fixed_delta_time());

        // Check if jump key is tapped
        bool jump_key_tapped = input::virtual_input::is_tapped("jump");

        // If jump is tapped, jump by adding jump force to the movement speed Y
        if (jump_key_tapped) movement_speed_.set_y(-constants.jump_force);

        // Add movement speed to position to make the character move, but apply over␣
↪time with delta time
        pos += movement_speed_ * timer::time_step::get_fixed_delta_time();

        // Do not let player to go out of the screen to top
        if (pos.y() <= 0.f) {
            pos.set_y(0.f);
            movement_speed_.set_y(0.f);
        }

        // Set the new position value
        registry.replace<transform::position_2d>(player_, pos);

        // ROTATION
        // Retrieve props of the player
        auto &props = registry.get<transform::properties>(player_);

        // Increase the rotation a little by applying delta time
        float new_rotation = props.rotation + constants.rotate_speed * timer::time_
↪step::get_fixed_delta_time();
```

(continues on next page)

(continued from previous page)

```
        // If jump button is tapped, reset rotation,
        // If rotation is higher than the max angle, set it to max angle
        if (jump_key_tapped)
            new_rotation = 0.f;
        else if (props.rotation > constants.max_angle)
            new_rotation = constants.max_angle;

        // Set the properties
        registry.replace<transform::properties>(player_, transform::properties{.
↪rotation = new_rotation});
    }

private:
    entt::entity player_;
    math::vec2f movement_speed_{0.f, 0.f};
};

// Name this system
REFL_AUTO (type(player_logic));

// Collision Logic System
class collision_logic final : public ecs::logic_update_system<collision_logic> {
public:
    collision_logic(entt::registry &registry, entt::entity player, bool &player_died)␣
↪noexcept : system(registry),
                                                                                ␣
↪            player_(player),
                                                                                ␣
↪            player_died_(player_died) {}
    // Update, this will be called every tick
    void update() noexcept final {
        auto &registry = entity_registry_;

        // Do not check anything if player is already dead
        if (player_died_) return;

        // Check collision
        check_player_pipe_collision(registry);
    }

private:
    // Loop all columns to check collisions between player and the pipes
    void check_player_pipe_collision(entt::registry &registry) {
        for (auto entity : registry.view<graphics::layer<3>>()) {
            // Check collision between player and a collidable object
            if (collisions::basic_collision_system::query_rect(registry, player_,␣
↪entity)) {
                // Mark player died as true
                player_died_ = true;
            }
        }
    }

    entt::entity player_;
    bool &player_died_;
};
```

(continues on next page)

(continued from previous page)

```
// Name this system
REFL_AUTO (type(collision_logic));

// Game Scene
class game_scene final : public scenes::base_scene {
public:
    game_scene(entt::registry &registry, ecs::system_manager &system_manager)␣
↪noexcept : base_scene(registry),

↪     system_manager_(system_manager) {
        // Set the constants that will be used in the program
        registry.set<flappy_bird_constants>();

        // Create everything
        create_background(registry);
        init_dynamic_objects(registry);
    }

    // Scene name
    std::string scene_name() noexcept final {
        return "game_scene";
    }

private:
    // Update the game every tick
    void update() noexcept final {
        // Check if player requested to start the game
        check_start_game_request();

        // Check if player died
        check_death();

        // Check if player requested reset after death
        check_reset_request();
    }

    // Check if start game is requested at the pause state
    void check_start_game_request() {
        // If game is not started yet and jump key is tapped
        if (!started_playing_ && input::virtual_input::is_tapped("jump")) {
            // Game starts, player started playing
            started_playing_ = true;
            resume_physics();
        }
    }

    // Check if player died
    void check_death() {
        // If player died, game over, and pause physics
        if (player_died_) {
            player_died_ = false;
            game_over_ = true;
            pause_physics();
        }
    }
```

(continues on next page)

(continued from previous page)

```
    // Check if reset is requested at game over state
    void check_reset_request() {
        // If game is over, and jump key is pressed, reset game
        if (game_over_ && input::virtual_input::is_tapped("jump")) reset_game();
    }

    // Initialize dynamic objects, this function is called at start and resets
    void init_dynamic_objects(entt::registry &registry) {
        create_columns(registry);

        // Create player
        auto player = create_player(registry);

        // Create logic systems
        create_logic_systems(player);

        // Reset state variables
        reset_state_variables();
    }

    // Create logic systems
    void create_logic_systems(entt::entity player) {
        system_manager_.create_system_rt<column_logic>();
        system_manager_.create_system_rt<player_logic>(player);
        system_manager_.create_system_rt<collision_logic>(player, player_died_);
    }

    // Reset state values
    void reset_state_variables() {
        started_playing_ = false;
        player_died_ = false;
        game_over_ = false;
    }

    // Pause physics
    void pause_physics() {
        system_manager_.disable_systems<column_logic, player_logic>();
    }

    // Resume physics
    void resume_physics() {
        system_manager_.enable_systems<column_logic, player_logic>();
    }

    // Destroy dynamic objects
    void destroy_dynamic_objects() {
        // Retrieve the collection of entities from the game scene
        auto view = entity_registry_.view<entt::tag<"dynamic"_hs>>();

        // Iterate the collection and destroy each entities
        entity_registry_.destroy(view.begin(), view.end());

        // Delete systems
        system_manager_.mark_systems<player_logic, collision_logic>();
    }

    // Reset game
```

(continues on next page)

(continued from previous page)

```
    void reset_game() {
        // Destroy all dynamic objects
        destroy_dynamic_objects();

        // Queue reset to reinitialize
        this->need_reset_ = true;
    }

    // Post update
    void post_update() noexcept final {
        // If reset is requested
        if (need_reset_) {
            // Reinitialize all these
            init_dynamic_objects(entity_registry_);
            need_reset_ = false;
        }
    }

    // System manager reference
    ecs::system_manager &system_manager_;

    // States
    bool started_playing_{false};
    bool player_died_{false};
    bool game_over_{false};
    bool need_reset_{false};
};

// Game world
struct flappy_bird_world : world::app {
    // Game entry point
    flappy_bird_world() noexcept {
        // Load the graphical system
        auto &graphic_system = system_manager_.create_system<sfml::graphic_system>();

        // Load the resources system
        entity_registry_.set<sfml::resources_system>(entity_registry_);

        // Load the input system with the window from the graphical system
        system_manager_.create_system<sfml::input_system>(graphic_system.get_
↪window());

        // Create virtual input system
        system_manager_.create_system<ecs::virtual_input_system>();

        // Define the buttons for the jump action
        input::virtual_input::create("jump",
                                     {input::key::space, input::key::w,␣
↪input::key::up},
                                     {input::mouse_button::left, input::mouse_
↪button::right});

        // Load the scenes manager
        auto &scene_manager = system_manager_.create_system<scenes::manager>();

        // Change the current_scene to "game_scene" by pushing it.
        scene_manager.change_scene(std::make_unique<game_scene>(entity_registry_,␣
↪system_manager_), true);
```

(continues on next page)

(continued from previous page)

```
    }
};

int main() {
    // Declare the world
    flappy_bird_world game;

    // Run the game
    return game.run();
}
```

1.3 Antara Gaming Modules

Table 2: Modules Tables

Name	Description	Dependencies	Header Only	Contains Script	Authors	Contributors
antara::config	Modules containing game configuration and utilities to load configuration	nlohmann_json			Roman Szterg-baum	
antara::core	Modules containing core utilities such as assets path, safe headers, etc…	EnTT refl-cpp			Roman Szterg-baum	
antara::math	Modules containing math utilities such as (vector2d, vector3d, etc)	refl-cpp			Roman Szterg-baum	
antara::input	Modules containing enums representing keyboard, mouse, etc…				Roman Szterg-baum	
antara::event	Module Containing all the common events that's will be used in games such as (key_pressed, key_release, etc…)	EnTT doom_meta refl-cpp antara::input antara::core			Roman Szterg-baum	
antara::timer	Module allowing us the manipulation of the game frame or retrieve the fixed delta time				Roman Szterg-baum	
antara::transform	Module Containing all the common transform components (position, rotation, …)	antara::event antara::math			Roman Szterg-baum	
antara::graphics	Module Containing all the common graphics components (color, layer, …)	antara::event			Roman Szterg-baum	
antara::geometry	Module Containing all the common geometry components (circle, square, rect…)	antara::event antara::graphics antara::transform			Roman Szterg-baum	
antara::audio	Module Containing all the common audio components (music, sound_effect)				Roman Szterg-baum	
antara::ecs	Module allowing us the manipulations of systems such as adding, updating, disabling, or removing systems (Contains a system manager, and system abtract class)	EnTT strong_type expected range_v3 doom_meta antara::timer antara::event antara::core			Roman Szterg-baum	
antara::animation2d	Module allowing us the 2D sprite sheet animation management	antara::ecs antara::graphics			Roman Szterg-baum	
antara::collisions	Module containing collisions detection functions/systems	antara::ecs antara::math antara::event			Roman Szterg-baum	
antara::resources	Module allowing us the manipulations of resources (textures, font, sounds)	antara::ecs			Roman Szterg-baum	
antara::scenes	Module allowing us the manipulation of scenes, possibility of adding, removing and changing a scene (Contains a Scene System Manager and an Abstract Scene class)	antara::ecs			Roman Szterg-baum	
antara::world	Module allowing the creation of application just by inheriting a simple class, contains all the necessary for the development of your	antara::config antara::core antara::ecs			Roman Szterg-baum	

CHAPTER

TWO

INDICES AND TABLES

- search

INDEX

A