Getting started with xygine

Introduction

Xygine is a small game framework based around SFML that I have developed over the past few years to create 2D games, which run on Windows, linux and macOS. It has become mature enough now, I think, that it's time to attempt to document the process of creating a small breakout style game, in order to (hopefully) better explain xygine's features and the functionality it provides. If you're reading this I'm going to assume some knowledge on your part of C++ as well as potentially SFML, and that you are here because you want a to shortcut skip creating much of the boilerplate code in favour of diving into creating a game. Let's begin!

Getting started

Xygine is a fully open-source project, licensed under the zlib license. The best way to get started is to clone the repository available on github, and build it yourself. The repository can be found at https://github.com/fallahn/xygine and the build instructions are on the wiki: https://github.com/fallahn/xygine/wiki/Building

Once you have configured, built and installed xygine for your platform of choice, it's time to create your first project! The repository contains a folder named cmake_template. This folder has all the files necessary to start a new project, including a default cmake file. Create a copy of this directory and rename it to something else such as 'breakout'. If you want to use the included cmake file then open that in your text editor of choice and edit the PROJECT_NAME variable to something more relevant too. If, like me, you prefer using an IDE you can open the cmake file directly if your IDE has support (such as Visual Studio 2017) or create a new project, adding the source and header files included in the template folder, and linking to the SFML and xygine libraries as appropriate (eg the correct debug/release versions). If you are taking the cmake route then the src and include subdirectories both contain a cmakelists.txt file, which will need to be updated if you add any new source files to your project. Once you have the project configured it should be possible to build and run it, which will open a console window (on Windows) and a blank game window titled "xyginext game". Pressing F1 will open a console window which displays any messages printed with the Logger class, and provides basic video and audio options. More information on the console can be found in the documentation https://github.com/fallahn/xygine/wiki

A closer look at the source files

The template project contains 3 sources files.

Main.cpp contains the entry point into the program, which instantiates a Game object and runs it. There's not much else to do here and will probably not be brought up again in the rest of the tutorial.

Game.cpp is where the magic begins to happen. The Game class inherits the xy::App class which is the root of any xygine game or application. The App class contains the render window instance, as well as manages events and system messages, making sure that they are dispatched correctly to the rest of the game. Documentation on App specific features (such as the message bus) can be found on the xygine wiki https://github.com/fallahn/xygine/wiki . Because the Game class inherits App it has accesses to these features through the App interface. The Game class is home to the StateStack - which will be covered later - and makes sure that events, messages and updates are correctly forwarded, as well as rendering the active states. Most important to note about the Game class, however, are the initialise() and finalise() functions, which are overrides of the App class functions. These are used to correctly create, configure and tidy up any extra game-wide resources which you may use throughout the lifetime of the application. They are called automatically on startup and shutdown and should be used if a game requires any third party libraries such as a physics system which requires specific initialisation and shutdown. The initialise function returns a bool - this should return false if any initialisation errors occur, particularly if initialising a third party library, as this will cause xygine to quit safely, rather than continue.

MyFirstState.cpp implements a simple State. States are used with xygine's StateStack class, which is optional but recommended. The StateStack is a concept which allows several states to be active at any one time, and controls how events and updates are passed down through the active states. A State object encapsulates a particular state of the game, such as the main menu, a gameplay state, or a pause menu. The advantage of a StateStack is the ability to keep existing states alive, such as a game state, while placing a pause state on top. States are rendered from the bottom of the stack upwards, and updated from the top down. This means that a game state at the bottom will be drawn first, with a menu (for example) in a pause state drawn on top. Conversely the pause state will receive updates and events such as keyboard and mouse first, and can 'consume' these events by returning false from the corresponding state functions handleEvent() and update(). This means that the menu remains responsive and active, while preventing the lower game state from being updated, effectively pausing it. Popping the the pause state from the top of the stack then allows the game state below to continue to execute as it starts to receive events and updates again. When used correctly this can be a very powerful and useful concept. For a more detailed look at the implementation of a StateStack I suggest reading 'SFML Game Development' by Jan Haller, Henrik Vogelius Hansson and Artur Moreira - a book which was very influential when I started creating xygine.

MyFirstState itself implements the interface of xy::State. The functions are similar to that of the Game class - handleEvent() which passes SFML events such as keyboard and mouse input

down to any game classes which need them, handleMessage() used to forward system messages to interested parties, update() which processes delta-time critical logic functions, and draw(), used to render the active state to the screen. The state also returns an identifier via stateID() which is used when registering the state with the StateStack in the Game::initialise() function. The specifics of this are covered later. Usually when creating a new project it is preferable to rename the MyFirstState class to something more coherent such as MenuState - but for this particular case I will leave it as it is.

Setting the Scene

MyFirstState contains an instance of a xy::Scene object. Scenes are used to encapsulate the Entity Component System or ECS of xygine, and multiple scenes can exist within a single state - for example one might be used to create the game area, and another to display the user interface. Entities are effectively no more than an integer ID, used to index arrays of components which are attached to them. As such they are very lightweight and easy to copy around, while still providing access to their attached components. Components are nothing more than data containers, usually structs, although some provide a selection of functions to create a cleaner interface, such as xy::Sprite. The key point is, however, that there is no logic held within a component, this is all performed by a series of Systems. A system class will look at all the active entities in a Scene and process each one based on the data held within its components. Xygine already has several component and system classes for often used tasks, but the ECS really shines when, later on, you create your own custom systems, and their corresponding components. To start with we'll use some of the existing components and systems to create a basic menu in MyFirstState. At the top of the cpp file include

```
xyginext/ecs/components/Transform.hpp
xyginext/ecs/components/Text.hpp
xyginext/ecs/components/Drawable.hpp

xyginext/ecs/systems/TextSystem.hpp
xyginext/ecs/systems/RenderSystem.hpp
```

A brief explanation:
The transform component is used by almost all entities to correctly place them in the Scene. There are very few cases where an entity will not need a transform - text and sprites cannot be drawn without one, buttons and UI components will not work, and audio emitters cannot be correctly placed. Transforms can also be parented to one another to form a 'scene graph'. More information can be found about this in the wiki https://github.com/fallahn/xygine/wiki

Text components are very similar to the Text class in SFML, although slightly modified to better fit the ECS. The interface is nearly the same, including functions such as setString() setFillColour() etc, and require an instance of sf::Font to render. For now a font can be added as

a member to MyFirstScene as m_font for example - later this will be replaced with a resource manager.

Drawable components are required for any entities which are rendered to the screen. They contain an sf::Vertex array which, in this case, will be updated by the TextSystem with data held in the Text component. Sprites and the SpriteSystem work in a similar way. Later, when creating custom systems, the Drawable component can be used to draw arbitrary shapes, having their vertex data updated via the custom system. Drawable components also have texture and shader properties, which allow SFML resources sf::Texture and sf::Shader to be applied to them. This will be covered later, and more information can also be found in the documentation. Drawables have another important feature: the depth property. The setDepth() and getDepth() accessors can be used to define the 'z depth' of a drawable, providing runtime sorting of visible drawables. Setting a depth with a higher value ensures that the drawable is drawn in front of those with a lower value. It's worth noting that there is no guarantee of order with drawables that have the same value as another - this might cause some visible flickering so in these cases a specific value should be set. The default value is 0.

Finally the RenderSystem is used by the Scene to sort and cull any visible entities which have a Drawable component, and render them to the screen.

Setting up the ECS

With the correct files included, the Scene can now be initialised. For convenience create a private member function in MyFirstState called createScene(), and call it from the MyFirstState constructor. Then define createScene()

```
void MyFirstState::createScene()
{
    //add the systems
    auto& messageBus = getContext().appInstance.getMessageBus();
    m_scene.addSystem<xy::TextSystem>(messageBus);
    m_scene.addSystem<xy::RenderSystem>(messageBus);

    //load resources
    m_font.loadFromFile("assets/fonts/my_lovely_font.ttf");

    //create an entity
    auto entity = m_scene.createEntity();
    entity.addComponent<xy::Transform>().setPosition(200.f, 200.f);
    entity.addComponent<xy::Text>(m_font).setString("Hello world!");
    entity.addComponent<xy::Drawable>();
}
```

So what have we just done? Firstly we've added the necessary systems to the scene. All systems require a reference to the xygine message bus so that they may subscribe to it. This is retrieved through the getContext() function, which is useful for accessing game/application properties such as the render window or message bus. xy::Scene::addSystem() is a templated function which takes the system type as a template parameter. The function parameters are all forwarded to the constructor of the system itself - multiple parameters can be passed, which is useful when creating custom systems. The default xygine systems, however, take only a reference to the message bus. Systems are updated and drawn in the order in which they are added to the scene, which is sometimes important. In this case we want the Text components to be updated before the RenderSystem draws them.

Next the font is loaded, which was added as a member to MyFirstState as m_font. This will eventually be replaced with a resource manager. Note that when loading the font there are no defaults supplied with xygine - you need to provide a path to your own. If the text is not visible then this is probably the first thing to check.

Finally a new entity is created. Entities are created through a Scene's factory function createEntity(). This is important because it ensures that the entity is correctly registered to its parent scene. The entity class is trivial, however, and instances can happily be overwritten to create new entities, while the existing one lives on. For example:

```
auto entity = m_scene.createEntity();
//add components to entity

entity = m_scene.createEntity();
//add other components to new entity
```

All entities will continue to live within a scene until explicitly destroyed, which will mark them for garbage collection:

```
m_scene.destroyEntity(entity);
```

This generally means that entities will exist until the end of the current frame.

Components are added to an entity with addComponent<T>(). Similarly to the Scene's addSystem() function, the entity's addComponent() function is templated, taking the component type as a type parameter. The function also forwards any parameters to the component constructor, such as in this example where a reference to the font is forwarded to the Text component. Entity::addComponent() will return a reference to the newly created component allowing properties to be immediately set, for example the string used in the Text component. A reference to a component can also be retrieved with entity.getComponent<T>(), allowing other component properties to be updated, or components to be copied elsewhere. Note that not all components are copyable, for example the Transform component is only moveable.

Building and running the project should now display the window as before, only with the words “Hello World!” in the top corner.

From here I encourage you to experiment with the entities and components - try updating the createScene() function to create 3 entities, each with a different text string: A title, one which says ‘Play’ and another which says ‘Quit’. Try updating different component properties such as the scale, position or rotation of the Transform component, or character size or fill colour of the Text component. When you have a basic menu laid out, move on to the next section.

Adding interactivity

To make the menu usable ideally the entities rendering the text will need to act as buttons when clicked on. Xygine includes a UIHitBox component and a UISystem for this purpose. At the top of the MyFirstState.cpp include

```
xyginext/ecs/components/UIHitBox.hpp
xyginext/ecs/systems/UISystem.hpp
```

Now in createScene() add a UISystem to the scene, right before adding the RenderSystem

```
m_scene.addSystem<xy::UISystem>(messageBus);
```

UISystems are slightly different from other systems - they require event input from the state, so that they know when to react to keyboard or mouse input. In MyFirstState::handleEvent() add the line

```
m_scene.getSystem<xy::UISystem>().handleEvent(evt);
```

With this set up it’s time to make the quit button do something. Hopefully in createScene() you should now have something like

```
entity = m_scene.createEntity();
entity.addComponent<xy::Transform>();
entity.addComponent<xy::Text>(m_font).setString(“Quit”);
entity.addComponent<xy::Drawable>();
```

To add interactivity to this entity add a UIHitBox component. This is used to define an area in the scene in which mouse events, such as button presses, perform a callback. The size of the text can be retrieved with the static function xy::Text::getLocalBounds();

```
entity.addComponent<xy::UIHitBox>().area =
    xy::Text::getLocalBounds(entity);
```

Now that the entity has an area defined for it, it needs to know what to do when it receives an event within the given area. These are done by callbacks registered with the UISystem. Callbacks are registered with a lambda expression or functor passed to UISystem::addMouseButtonCallback() which returns an integer ID. The ID is useful because it means a single callback function can be assigned via its ID to multiple components, should those components wish to perform the same thing.

```
auto callbackID =
m_scene.getSystem<xy::UISystem>().addMouseButtonCallback(
    [&](xy::Entity e, sf::Uint64 flags)
    {
        if(flags & xy::UISystem::LeftMouse)
        {
            getContext().appInstance.quit();
        }
    });
```

This looks a little wordy, so let's break it down. A reference to the UISystem can be retrieved from the scene with getSystem<T>(). Then a lambda expression is passed to addMouseButtonCallback(). The lambda signature for mouse button callbacks passes in a copy of the entity which is executing the callback - that is, the entity which has the UIHitBox component attached to it, and a copy of the event flags. Event flags can be used to check which mouse button was pressed. In this case we're looking for the left mouse button, and if it is true the game is quit. (xy::App::quit() is the preferred method, as it makes sure everything is finalised properly before closing the game). Other available callbacks, their signatures and their flags are detailed in the documentation. https://github.com/fallahn/xygine/wiki

The result of registering the callback is returned as callbackID. To associate this with our hitbox (and potentially any others) it needs to be added to the hitbox callback array.

```
entity.getComponent<xy::UIHitBox>().callbacks[xy::UIHitBox::MouseUp] =
callbackID;
```

Now whenever the hitbox detects a mouse button up event the callback with callbackID is performed. In this case the button is checked to see if it is the left button, and if it is the game is quit.

Compile and run the project. Clicking on the text which says ‘Quit’ should now close the window.

From here, using the documentation as reference, experiment with the UISystem and its callbacks. As well as mouse events try handling keyboard events. Mouse move events can be used to trigger callbacks which highlight the selected text for example.