Progeny Engine: Generation 1 Documentation

March 16, 2019

1 Introduction

This documentation is meant to serve as a means to better understand the inner workings of the Progeny Engine. This documentation may also serve as a means to understand the workings of the now legacy "Core Engine" from which the code and concepts the Progeny Engine is derived. Any application made with the Progeny Engine can be logically broken up into three parts: 1) Interface Implementation; 2) Engine Implementation; 3) User Implementation. The most common part anyone using the engine would be working with would be the third part, User Implementation; but, for the sake of everyone involved, this documentation will cover all three. Before we get too into the workings of the engine it could be useful to understand it's history.

1.1 A Little History Lesson

Some of the code within the Progeny Engine dates back to late 2014 and possibly even 2013. Concepts and experience were gathered from an older engine created in late 2013 and early 2014. At this time there was no other goal than to create an engine and learn from it. In around 2015 the code was transfered to another system and optimize more, by this point the engine was unofficially called the "Core Engine". The Core Engine had enough functionality for a simple game. A demo titlescreen for an imaginary game called "Free Runner" was created using the engine as a prank and sortly after, for no unrelated cause, it was deemed logical that the development of a game alongside the engine would help determine flaws in it's design; this was the begining of the game now called "Generic Jumper". Eventually the game moved to source version control and for the lack of a more interesting name "Core Engine" ended up sticking. After the release it was obvious that the engine had many flaws in design. Some issues with the engine were solved during the development of Generic Jumper with the use of temporary, unsatisfying, patches. Generic Jumper released on August 31, 2018 and the Core Engine was shelved and it's derivative

born: Progeny Engine Generation 1. Named in such as a means to describe it's evolution from the Core Engine and that the new engine is meant to tackle issues found in the older engine's design.

2 The Rough Details

This section of the documentation is to give the reader an idea of what different parts of the engine do in a vague over simplistic way. It is meant to help someone gain a slight understanding before they dive neck deep into the more complicated text. It also provides a means for someone wanting to know if this is the right engine for them to use. If there were any section in this documentation that would be recommeneded above all others it would probably be this one as this section will help the reader create a basic understanding of the engine's structure.

2.1 Interface Implementation

Some naive developers might strafe right into developing an engine for a game with the main engine code calling methods to an external graphics library. Of course, there are reasons to why you would like to do that for a project yourself but if you have no idea what systems your engine will be running on it is a good idea to have some way to seperate engine logic from system dependent logic (such as media layer calls).

2.2 Engine Implementation

The Engine Implementation in a perfect world would not have to be touched by many individuals. Most of what will be covered in this documentation will not be on how to write the Engine Implementation (as it is already written and if changed makes the rest of this documentation absolete) but how the Engine Implementation interacts with the Interface Implementation and the User Implementation. In a nutshell (perhaps a very too obvious nutshell) the Egine Implementation is the core implementation of the engine that defines the behaviours of the classes it presents to be used by the User Implementation.

2.3 User Implementation

This is what makes a game different from all the other games. It is the logic behind the game. Not much else to say here.

2.4 Game States

Every game will have at least one class, a 'game state', that inherites the GameState class. This class implements virtual functions that are defined by the GameState class

and are called at various points during the program's lifetime. Game states are the logical seperations in the game's logic and so it will be natural to have multiple game states per game. Think of a game you have played. It will have distinctive seperations in gameplay, perhaps it had a: titlescreen; save file select screen; cutscenes; and, most probably, some sort of main interactive state. Game states are not limited to these examples but can consist of loading screens and other non interactive parts of the game; it's really up to you!

Another important thing to note about game states is that they inherit the StateManipulator class through the GameState class. This give the game states control over the flow between game states. The GameState class gives the user virtual functions to implement the behaviour of various points during a game states lifetime where the StateManipulator class gives the user a way to cause change to the game states lifetime; this will be discussed in more detail in a later section.

2.5 Game Objects

Game Objects are the building blocks of everything the player interacts with. If game states were the stage of your game then your game objects would be the props, the backdrops, the curtains, the actors, etc. It is possible to build a game using the engine without the use of the GameObject class but it would be like re-inventing the wheel when someone already invented the car! To better exaplain without the use of a super vague analogy: A game object is made up of different smaller parts where each component has some sort of effect on how the player interacts with it. These smaller parts are called components and they inherite the ObjectComponent class.

2.5.1 Object Components

There is currently 5 ObjectComponent classes: TransformComponent; SpriteComponent; RigidBodyComponent; ColliderComponent; and the Controller ObjectComponent class. The last one is named differently simply because it has a special use and is handled a bit differently then the rest. If one would want they would be able to add more ObjectComponent classes but this would require the use of modifying the Enigne Implementation so that the Engine would know what to do with the newly added component.

To go over in brief, the TransformComponent represents an object's position, rotation, and an identifyer; the SpriteComponent represents the image (sprite) to use along with some associated data; the RigidBodyComponent represents the physical properities the object has such as mass and restitution; the ColliderComponent is the shape of the object itself; and the Controller ObjectComponent is what defines the object's behaviour (such as how it moves around the screen or what happens when it collides with some).

2.6 Listener Classes

EventListener, TriggerListener

2.7 Controller Classes

2.8 ...

3 A Guide to the Basics

This section is meant to be a guide to starting and maintaining a project. It goes over good practices to follow and tips to efficiently use the engine.

3.1 Where to Start

This subsection goes over the setup of a basic project. In most cases the code provided in this subsection will be suffice for most games you would want to create using this engine.

3.1.1 Installation

If you are using the interface implementations provided by the engine then you will be required to install a few extra libraries; if not then you will have to build the implementations yourself (this is not recommend for anyone not familiar with the engine). How to implement the interfaces will not be discussed here; information on doing so can be found in "A Guide to the Not-So-Basics" section.

3.1.2 Sanity Check

3.1.3 main.cpp

3.1.4 The First Game State

3.2 Where to Continue

This subsection builds off of the previous subsections code although you really only will need a basic project setup. We will first start off with designing and managing GameObjects.

3.2.1 Designing GameObjects

GameObjects are designed to take advantage of C++'s polymorphism features. This is to allow the reuse of code for different GameObjects. An example of this could be to have all GameObjects that have an animated sprite inherit their main implementation from another GameObject that implements the animated feature. It is also important

to know just how GameObjects are processed by the engine to make full use of them. First we will create a new header file named "mob.h". In this header we will include "src/engine/GameObject.h" and define class named "Mob" that will inherit the class "GameObject" publicly. The source should look like this:

3.2.2 Behaviour with Controllers

Instead of adding changes in a GameObject's data through GameState functions you can do so through Controllers.

3.2.3 Organizing GameStates

3.2.4 Events and User Input

go over enums and static classes that call EventInterface::pushEvent(..) so that implementation of the data pointers can be transparent to users. This is important for larger projects with multiple maintainers as forces the implementation to be in one class. For example if one of the pointers is dynamically allocated then having a user do eventInterface.pushEvent(..., new DataObject(), ...) feels unsafe because they're passing ownership to something that might feel unknown. So defining functions that push events in the same class(namespace) means that the new and delete will be in the same namespace aswell... and the users can simply call Mob::pushJumpEvent(); to push a jump event. (This can be seen in Walkway for the dialogue system since it's passing pointers to std::string's.)

3.2.5 User Event Practices

3.2.6 Triggers and Collision Detection

3.2.7 Persistence with Strings

3.2.8 Audio and FX

4 A Guide to the Not-So-Basics

This section is meant to be a guide to the modifiable elements of the engine that are outside the User Implementation side of things. For example, adding a new GameObject-Component or anything else requiring you to modify/add/remove engine code. View the subsections below to see if what you would like to do is covered, otherwise email us at query@asphe.games to inquire more!

4.1 Interfaces

This subsection is meant to go over the basic contracts defined by each Interface class. Details on what each paramater means and what they are expected to do are discussed

below for the sake of both users and implementors that want to know how an interface call is supposed to behave.

4.1.1 File Interface

4.1.2 Event Interface

4.1.3 Graphics Interface

---

void
GraphicsInterface::renderClear(
);

Clear the current rendering target.

1. Sets each pixel of the rendering target to the colour set by *GraphicsInterface::setDrawColour(uint32_t colour)*.

---

void
GraphicsInterface::basicRenderCopy(
int32_t textureID,
int32_t destX,
int32_t destY
);

Render the entirety of a texture on to the current rendering target.

1. *textureID* is the ID of the texture that is to be rendered.
2. *destX* is the X-Coordinate where the upper left corner of the texture will apear on the rendering target.
3. *destY* is the Y-Coordinate where the upper left corner of the texture will apear on the rendering target.
4. The texture will be rendered to the rendering target to a 1:1 pixel ratio.
5. If the function is passed an invalid texture ID the rendering target will be unaffected.
6. *destX* and *destY* may be any value; any area of the texture that overlaps with the rendering target will still be rendered.

---

void
GraphicsInterface::renderCopy(
int32_t textureID,
Rectangle const * const sourceRect,
Rectangle const * const destinationRect
);

Render a specified area of a texture on to a specified area of the rendering target.

1. *textureID* is the ID of the texture that is to be rendered.
2. *sourceRectangle* is the 'specified area' of the texture to be rendered to the rendering target.
3. *destinationRectangle* is the 'specified area' of the rendering target that will be rendered too.
4. The 'specified area' represented by *sourceRectangle* will be scaled to match the size of the 'specified area' represented by *destinationSource*.
5. If *sourceRectangle* is null then use the entire texture. (?)
6. If *destinationRectangle* is null then use the entire area of the rendering target as the destination. (?)

---

void
GraphicsInterface::setWindowSize(
uint32_t width,
uint32_t height
);

Changes the window's width and height.

1. When called before *GraphicsInterface::createWindow()* this function is to set the window size in pixels by mapping viewport pixels to display pixels with a 1:1 ratio.
2. When called after *GraphicsInterface::createWindow()* this function is to set the window size in pixels without changing the viewports resolution.

4.1.4 Audio Interface

4.2 Adding GameObjectComponents

4.3 ...

5 Persistence

5.1 Macros

STRMAP
STRSET
STRGET
STRKILL
VARSET
VARGET
VARKILL

6 Inherited Classes

6.1 Controller.h

6.2 EventListener.h

6.3 GameObject.h

6.4 GameState.h

6.5 Packageable.h

6.6 StateManipulator.h

6.7 TriggerListener.h

7 Packages

7.1 GuiPackage::Label()

8 Generic

8.1 Point.h

8.2 Rectangle.h

8.3 Vector.h