TEAM | YARBUS

Presents

Treading Lightly

Table of Contents

Overview

Project

Tread Lightly is a 2D poetic experience of a character treading their way through a dark and perilous world. They are faced with environmental obstacles like wind, rain, fire, as well as a personal obstacle that appears as a motif of a memory they are haunted by. To get through the darkness, they utilize a special lantern that allows them to see a light and beautiful world through it.

For this project we are using C++ and Zilch as our scripting language. For our standard libraries, we are using the STD, glm for math library, and GLFW for our window. We are using GLFW so that our game is cross platform compatible.

Engine

The Cinder Engine is a component based engine that uses a sign-up system for the components to communicate with each system. This allows us higher performance as compared to calling update on every component as only components which are being used are called. The engine has pointers to each system allowing base communication between every system in the engine.

Graphics

The graphics of the engine will use GLFW in combination with custom OpenGL shades. We are using OpenGL4.2 because of its powerful compute shaders which allow us to have millions of particles on screen at a decent frame rate. This allows us to create powerful visual effects to emulate fire and in tandem with thermodynamics system will create a pseudo-realistic fire effect. Other graphics such as the Background, Level Objects, UI, and Menus will use a Sprite component. This component renders 2D art to the screen using .png format. All sprites and their textures, shaders are stored and referenced from the Resource Manager.

Most of our visuals are procedurally generated. We are using a combination of L-System and Fractal Mathematics to generate trees, terrain, grass, other vegetation.

Physics/Thermodynamics

Our physics and thermodynamics systems will work together to create the movements of fire as it grows. To do this we use simple colliders such as point, circle, and polygon collision to create our environments which then interface with the thermodynamics systems to create a flow of heat. The majority of our levels use a terrain collider which is actually a collection of polygon colliders in a single component. In creating these we can create large environments by which our fire will spread. Thermodynamics will be used to support this growth across the terrain and objects in the environment.

We're using a 2D thermodynamics system for simulating heat flow around the level. Levels are mapped to grids which store temperature, density and material data. Volatile objects register themselves to the system via components.

Heat flow around the map is used to generate air currents which, in turn, influence the flow and weather in the game. This allows us to have a completely dynamic fire propagation system that is subject to weather effects.The system is based on the paper "Real-Time Fluid Dynamics for Games" by Jos Stam, and uses a real-time Navier-Stokes solver algorithm.

Audio

The Audio system is built around the FMOD Studio Low Level API. We have basic features like play, pause, mute, stop, pan, fade etc. We also have channel groups to mix different segments of audio. We are also using positional audio to map out the audio in a stereo spectrum according to where the audio source is situated in the level.

We have successfully implemented microphone input into the engine. The microphone input has a variety of features. It has the capability to detect the level of the incoming input. It can also detect what frequency range the input audio lies in, which in turn lets us detect musical notes. We can also filter out frequencies of the microphone input. For example if we want to detect only the blowing of air, we can block out all the frequencies other than the 20 - 150 Hertz range.

We are using a number of DSPs (Digital Signal Processors) from their libraries to create some of our desired effects. This allows us to modify all sounds being played in the game such as how the music is affected by the environment. As an example, we can add reverb to the music and sound effects when in a large building to make the audio reflect the environment. And also pass in Low Pass Filters in the signal chain to achieve an underwater effect. We can dynamically interpolate between the cutoff frequency and the resonance of the filters. We also have a parametric EQ for changing the tone of the sound (To make it sound dark or bright). We can interpolate the Bandwidth, Frequency and the Gain of the Parametric EQ. Our audio system is also using the feature of procedural audio generation using wave oscillators to create a wind sound effect. This effect can be dynamically changed when we needed.

Tools & Development

We plan to use the Zero Editor as our level editor as we will be able to quickly and efficiently create levels in Zero engine and then import them into our engine. The use of Zilch as our scripting language should also increase development time as any prototypes should be easily convertible from the Zero engine with only minor changed. We anticipate that using a scripting language will help us in the long run as developing code in C++ takes a much than a language like Zilch.

We also plan to have a small number of specific system tools to easily modify stats in the engine. Specifically using Anttweakbar to modify audio setting and tweaking variables in the game so that we can quickly modify levels without any major revisions via the use of the Zero Editor.

Graphics Implementation

Implemented API

The graphics system will go through GLFW3 and use some of the features from OpenGL 4.2 Using GLFW3 allows us a large degree of freedom in the development of our game because not only will our game be cross-platform, but it also gives us a nice interface of OS specific callback, and basic OpenGL initialization functionality.

Component Interface

Every graphical aspect of the game runs through serializable components. Each component holds their related set of data and functions required to render their specific component type. This is done by all graphics components inheriting from a base graphics component with a virtual Draw function. The Graphics system will then call the virtual Draw function to draw the related graphics function every frame.

Resource Manager

Shaders

The graphics system will load compiled shaders into the Resource manager which will then be used from the central Graphics Pipeline. This will allow us to keep track and use all of our shaders across levels and on any object. The specific graphics components will then hold the data to use specific shaders which then become serializable by name.

Textures

The graphics system will load textures into the resource manager. The loaded textures will then be referenced by the central graphic pipeline to enable all objects to share the same textures. These textures can then be scaled appropriately to each object related to the data on each object. This will allow us to modify how textures look on each sprite and have them scaled to each objects relative dimensions.

Behavior Implementation

Ember has limited AI behaviors as the majority of the game are physics interactions. We foresee several other core interactions between our player and game world which will run through the thermodynamic system but will also be mentioned here.

Fire and Heat, the player is fire and as such his degree of movement is much greater in hot burning air than it is were cool. With this in mind we plan to have the player's movement severely restricted when outside of their flames.

This is done by referencing the TemperatureMap held by the thermodynamics system. By getting the temperature the grid in which the player is in we can modify the player controller to affect how quickly the player can move and how much gravity affects the player.

Fire and Smoke, in closed environments the player will burn fuel which creates a lot of heat and smoke. The heat will initially help the player move, however, if enough smoke builds up the player will become sluggish similar to if they were in cool air, but it is exponentially more effective against eh player's movement.

This is done by referencing the SmokeMap which gets the density of how much smoke is in the cell the player is in. Using this value we can then modify the player controller to affect how quickly the player can move and how much gravity affects the player.

Fire and Water, Water is our first hazard and upon contact the player will be extinguished.

In the case of non-fluid water we will use basic collision of circle to circle detection to determine when a water droplet, or stream of water hits a player. When this happens we will decrease the heat of the player causing the player to move more slowly and have less glow.

Physics Implementation

For our physics implementation we are using Improved Euler method for updating our objects. We do not plan on using any special partitioning because our game should have no more than a dozen different colliding bodies per level. For our colliders we want to implement several different colliders so that we can create terrain for our player to move along the ground as some other more complex shapes. Colliders to implement: Point Collision Detection/Resolution (D/R), Circle Collision (D/R), and convex Polygon (D/R). To help us implement this we are looking at Randy Gaul's tutorials and example physics engine to quickly understand the fundamentals of a good physics engine.

Audio Implementation

Ember's audio uses FMOD Studio for its audio. It uses a number of advanced features such as Low-pass filter, High-pass filter, Reverb, Parametric EQ.

Multiplayer Implementation

Our game is a single player Poetic Experience so this section is somewhat out of place, but my OCD nature prevents me from excluding it.

Coding Methods

General Coding Guidelines

We do not plan to have a hard set of rules for coding format, but there are a few musts when it comes to organizing code. You **MUST** use a **.h** and **.cpp** file. For your functions and/or classes use declarations in the h files and implementation in the .cpp file. The only exception to this is template classes or functions in which case include a .tpp file at the end of your .h file. You **MUST** include the precompiled header at the very top of your cpp files. In **.h** files you should not be including any other files, instead use a forward declaration and include the **.h** directly after the precompiled header in the **.cpp** file. Also, remember to fill out the file information at the top of the file name, your name, description, etc…

It is recommended that you have a line of comment above all of your functions which have ambiguous names or convoluted logic. Any code which is specifically complex in the area of its logic or control path should by heavily comments so that fellow co-workers can look through your code.

Debugging

Cinder Engine Debug Features

Trace, is a quick way to write to the console to tell the use a specific system or object has been created. This is meant to be used for initializing engine system and other major events such as Level loaded, initialized.

Assert/ErrorIf, are expressions which stops the programs, writes to the console and error message, and also pops up a windows message box reading out the message. These should be used for Critical checks which will prevent the engine from running. They may also be used for debug purposed for their debug functionality.

Console Output, the console can be used the standard std::cout. Also if you want to display certain text in specific colors the programmer can use "std::cout << CinderConsole.Color" to change the color which is output into the console. These are to be used to distinguish what output is of highest importance and can be used to differentiate output for debug and general programming.

Tools

Microsoft Visual Studio 2013

Although we are making a cross-platform compatible game we will be using Visual studio 2013. We are also using several C++11 features so you must have a compatible compiler.

Zilch Debugger

We are using Zilch as a scripting language and alongside this we will be using the Zilch debugger to step though our zilch code as it executes to determine what specific parts of the engine are failing. This could potentially be very useful because of the possibility of objects being destroyed and Zilch scripts trying to access them.

Appendix A

Zero Engine

We are using the Zero engine to build any of our level which are not procedurally generated, aka Menus, Credits, End Screen, etc. A majority of our content is procedurally generated so we will likely not need to use this very often, but in the case of testing specific events it could be useful and because our serialization can read Level files from Zero engine this is particularly useful.