The Enemy Within (Working Title), Technical Design Document

David Robertson, Thomas Hope, Davide Passaniti

Contents

Overview

Concept

The game is about cells growing and changing state. The player has to manage the growth of cells on a hex grid, cells will move autonomously if the player doesn't take action. Each cell has internal state that changes based on adjacent cells.

Technical Goals

The code team will be implementing the game in C++ using cross platform libraries. The code should be tailored specifically to our game, object orientated and minimal. Given the limited time the focus will be on simplicity and functionality over 'advanced' code.

Technical Risks

- Not having core gameplay features implemented
- Other members of the team being held back by slow engine progress
- Programmers having to be assist in testing game assets
- Could be harder to debug
- Might not build for target platform
- spending lots of time on engine programming, not enough time to polish
- 'infinite' grid required memory allocation = crashes (mitigate with modern c++ features)
- Bugs

System Requirements

Target platform is Android Tablets. The application will be compiled and tested with the AndroidSDK, NDK and android studio.

Technology

- C++
- OpenGLES
- SDL2
- SDL2_mixer
- SDL2_image

Programming Team

- David Robertson
- Davide Passaniti
- Thomas Hope

Feature List

Technical

- splash screen
- main menu
- optons menu
- Touch input, single touches
- Drag navigation
- Pinch to zoom
- hex grid
- “infinite” grid (memory allocation)
- scene culling (don’t render the entire world)
- nodes preform checks on adjacent nodes
- nodes change their state + spawn based on user input
- minimal path finding / line drawing on the grid

Art

- loading png images
- static images
- rotated / scaled images

- animation from sprite sheets
- randomised image selection
- randomised animation
- blending between animations
- transparency
- colour tinting

Audio

- loading wav files
- loading mp3s
- play audio once
- volume control
- loop audio
- randomly select sound
- pitch shifting
- time shifting

Research

Feasibility

- A team last year did it

Current Progress

- david is working on building an engine for OpenGL
- davide is building openGLES projects for android
- Thomas is iterating on prototypes

Challenges identified

An easy to use pipeline for the visual and audio artists will be essential to ensuring fast iterations and all members of the team are able to fully bring their skills to bear on the project.

Implementation

Source Control

We will be using Git source control. The engine is currently stored in a private repository on Github.

Testing Tool

While the engine is under development it would be very valuable to have some kind of instant feedback tool for the other members of the team. Rather than having to wait to have a programmer to integrate some new asset into the game they could test and iterate quickly on designs in their own time. The testing tool could show previews of animations under certain game conditions or how the manually created art would look alongside some procedural art or shader. It could also allow for the testing of audio assets, randomised events, or asset pools.

Infinite grid

The game will consist of a 2d hex grid which the player will be able to grow their cells across, hopefully infinitely. While it is impossible to have a truly infinite grid without infinite memory it is possible to create a game world so large that no user will conceivably reach the boundary. Game with such worlds already exist, the most popular of which is Minecraft. Minecraft's world is three dimensional, and infinite in two dimensions. The game is divided into 'chunks', each chunk being dynamically created as required and unloaded when no longer needed.

Will likely use STL containers since they are robust and well documented, the reference below makes use of a `std::map` to store chunks of the grid. Each chunk can be sorted within it's container for easy lookup. When checking the contents of some grid coordinate, first look in the container to see if the chunk with that coord exists. If it does search within the chunk to find the coord, if not then allocate the chunk.

The allocating and linking of chunks of memory should be handled within a game board class. Callers can then use simple `set(x, y)` and `get(x, y)` functions without having to worry about the implementation. When it comes to rendering the board a check will have to be done for each chunk to see if it's on screen so only chunks that are actually visible are drawn.

sources:

- https://github.com/gummikana/infinite_grid.cpp
- http://www.redblobgames.com/grids/hexagons/

Type 3 engine on windows desktop with openGL ES

Type3 engine was initially built as an SDL/OpenGL based technology with Visual Studio, since in the early stages of design it was not clear what platform we would be developing for. When the decision was made to create an android game, it was clear that we would have to switch the graphics API to OpenGL ES, and therefore it was of utmost importance to test whether or not our current code could be adapted for use with embedded systems or if we would have to rebuild everything from scratch. The decision was then made to create a quick test branch of the current code substituting GL initialisation code with GLES code.

The first difficulty encountered was that the GL extension wrangler library (GLEW) used to load the API's extensions did not offer support for GLES (probably because there is no reason anyone would want to write a desktop application based on ES, apart from testing purposes), and therefore we would have to load them manually. To do this, we used SDL's `SDL_opengles2.h` header, which provides all the function definitions, macros and typedefs useful for GLES v2.x, and SDL's `SDL_GL_GetProcAddress()` function, used to grab and assign the API function pointers.

The following is an example of the implementation: 1. suppose our code requires the use of the function `glCreateProgram()` 2. make sure `SDL_opengles2.h` is being included in the file where we want to use the function 3. looking up the definition in `SDL_opengles2.h`, identify the types for the function's return value and input parameters (in this case `Gluint` and `void` respectively) 4. create a typedef similar to this:

```
typedef GLuint(APIENTRY * GL_CreateProgram_Func)(void)
```

(APIENTRY resolves to ___stdcall)

5. Create a variable with the function's name using the new typedef, making sure the scope is local to where the function has to be called

```
GL_CreateProgram_Func glCreateProgram = NULL
```

6. Finally, assign the function's address to the variable using

```
glCreateProgram = (GL_CreateProgram_Func)SDL_GL_GetProcAddress("glCreateProgram")
```

7. Now `glCreateProgam()` can be used as normal in that scope.

This process has to be repeated for every required function. Since most of OpenGL's functions we were using had a GLES counterpart with the same or similar name, the engine's structure only suffered very minor changes apart from substituting GLEW includes and initialisation calls with the process described above. The only other addition worth mentioning is the need to specify a GLES context when creating the window:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,SDL_GL_CONTEXT_PROFILE_ES);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 2);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 0);
```

In the end we were able to have a window with a GLES context running on Windows, and the only non reusable code ended up being the few lines of test shader written in desktop hlsl.

Compiling SDL for Android

After confirming our technology could be adapted to the new platform, we needed to put it to practice. SDL is written in c and our engine in c++, so to compile it and run it on an android device the Native Development Kit (NDK) has to be used. The first step was therefore to compile the example android project provided with the SDL 2.x source. This is an Eclipse/androidSDK project that contains all the Java files required to start the application and interface with SDL's function calls and any c/c++ files the user wants to add. For a number of unfortunate coincidences however, this proved to be a harder task than it seemed. In fact, android sdk support for Eclipse has recently been discontinued in favor of Android Studio, which means the example project had to be ported to the new IDE unless we wanted to try and find an old version of Eclipse with androidSDK and develop in an obsolete environment. Other than the minor annoyances of changing IDE however, the biggest problem was that since Android Studio is a relatively new tool, its current NDK support is not complete and anything more than compiling a single c/c++ file with native function definitions in it requires one of the following things:

- Either downloading an experimental plugin for Gradle (the build automation system used by Android Studio) with NDK support, which is still being worked on in these months
- Or tricking Gradle into thinking your project has no native code by changing the build script, and then running NDK build tools manually from the command line, effectively compiling your java project and c/c++ files in two separate steps.

After spending a lot of time trying to automate the whole process with the experimental plugin and struggling with its DSL (the language use to write

gradle scripts) syntax changing week by week, we decided to temporary settle for option two. The following list describes the process with which we managed to get the SDL android project to run on Android Studio's emulator (Nexus 5):

1. Download Android Studio, the android NDK and the SDL2 source code.
2. Install the sdk platform and tools for the target platform you want to develop for(NDK supports up to 21, we used 18 in this test). This is done via the SDK-manager tool that comes with the IDE.
3. Set the environment path variable to where the ndk folder is to be able to call build commands later on (on windows 10: control panel > system and security > system > advanced system settings > environment variables).
4. Start android studio and choose "import project (eclipse, gradle,...)".
5. Select the "android-project" from the SDL source folder.
6. Copy the SDL source folder in `YourNewProject\app\src\main\jni`.
7. Create a main.cpp file in `YourNewProject\app\src\main\jni\src` and add your code(in our test: initialise SDL, create a GLES window, draw a triangle with a shader, wait for input to quit).

Then make the following changes to the following files (using api 18 for examples)
1. `YourNewProject\app\src\main\jni\Android.mk`: add `APP_PLATFORM := android-18`

2. `YourNewProject\app\src\main\jni\src\Android.mk`: change `yourSourceHere.c` to `main.cpp` and make sure `SDL_PATH` is the correct relative (or absolute) path to your SDL source folder

3. `C:\Users\<USER>\.gradle`: create a file named "gradle.properties" and add `android.useDeprecatedNdk=true` in it

4. `YourNewProject\local.properties`: add `ndk.dir=<path to your ndk folder>`

5. `YourNewProject\app\bulid.gradle`: change `compileSdkVersion`, `minSdkVersion` and `targetSdkVersion` to `18` inside defaultConfig scope add: ~ ndk { moduleName 'main' } ~ inside android scope add: ~ sourceSets.main { jni.srcDirs = [] jniLibs.srcDir 'src/main/libs' } ~ This is where we tell gradle we don't have native code to compile by clearing the native source directories paths

6. In android studio's terminal (or in the command prompt) move to `YourNewProject\app\src\main\jni` and run the "ndk-build" command. This will build SDL and the cpp files previously added.

7. Finally, run the gradle build and start the correct emulator (in our test a nexus5 with api 23).

With this process we were able to run an OpenGL ES shader on android without writing any Java code thanks to SDL. The downside, other than the build process being split in two steps, is that to prevent gradle from trying and failing to build our native code we cannot see our c/c++ files in android studio's project tree.

Engine Overview

Purpose

The purpose of this piece of software is to enable the development of the game "The Enemy Within" via providing adequate tools for developing this title.

Scope

To create a framework that will provide the required tools for making a 2D game.

References

Making Games With Ben - https://www.youtube.com/user/makinggameswithben

Overview

This framework will provide the following tools for game creation * A camera class * Error reporting * An input manager * A resource manager and the loading of resources * A sprite class * Sprite Batching * Timing commands

Design

Camera Class

The camera class consists of seven functions, however five of these are either "getter" or "setter" functions so they simply update or return a variable value. The remaining two functions consist of an initialisation function which simply sets up the camera class by initialising all of the variables and giving them base values.

The final function is the update function which does the heavy lifting of the class, when called it will check if the camera needs to be updated or not. If it does it will translate the camera's matrix to the correct position and then scale it accordingly.

Error Reporting

This will simply output an error string and check for the users's input before exiting the program.

Input Manager

The input manager consists of 3 functions and an unordered map. Two of the functions are very simple; they either set the value of the key to true or false in the unordered map. In doing this however, if they value of the key has not been added to the map yet, it is now added with the value it has been set to. The "is key pressed function", checks to see if the value of the key that is being checked is in the unordered map. If it is not then false is returned. If the key value is in the unordered map, the Boolean value of that key is checked and if it is true, true is returned, otherwise false is returned.

Resource Manager

The resource manager consists of one function and a texture cache. The function "Get Texture" simply hands the file path onto the texture cache, which in turn will check to see if the file is contained in the texture cache, if it is then it will return the texture. If it is not then the texture will be loaded into the cache from the file path and returned.

Sprite Class

The sprite class simply contains two functions, the initialisation function which sets up all the base variables for the new sprite and loads the desired texture from the texture cache.

Sprite Batching

- The Sprite Batching class consists of a "Glyph" struct, a "GlyphSortType" enum class, a "RenderBatch" class, five public functions and six private functions.
- The "GlyphSortType" simply defines what sorting method will be used in the sorting functions.
- The "Glyph" is used to store the texture, the depth and a vertex for each corner of the sprite.
- The "RenderBatch" class contains only one public function and three variables, the function simply initialises the render batch class.
- The initialisation function of the sprite batch class simply calls the create vertex array function.
- The begin function sets the sort type to TEXTURE, and then clears out all of the render batches and glyphs.

- The end function sorts all of the glyphs and creates all of the render batches.
- The draw function takes in all of the information about a glyph and creates a new one using this information. Then it pushes the glyph into the glyph vector.
- The render batch function firstly binds the vertex array and draws all of the render batches. It then unbinds the vertex array.
- The create render batches function checks to see if the glyphs vector is empty, if it is then the function returns. If it is not then the function continues. Runs through all of the glyphs in the glyph vector and adds them to the corresponding render batch, depending on what texture they are using.
- The create vertex arrays function simply generates vertex arrays and makes sure that there is a value for the vertex array and the vertex buffer.
- The sort Glyphs function checks what the value of sort type is, and then uses std stable sort to sort the vector of glyphs accordingly.
- The three compare functions are simply used to compare the values of two inputs. These are used by the std stable sort function.

Timing Commands

The `FpsLimiter` Class consists of four public functions and one private function. The initialisation function simply sets up the variables with base values and sets the maxFPS to the desired value. The set max fps function simply sets the max fps value to the value passed into the function. The begin function sets the start ticks value to the value returned from the `SDL_GetTicks()` function. The end function firstly calculates the fps by calling the calculate fps function and then gets the frame ticks by taking the start ticks value away from the current value returned by `SDL_GetTicks()`. It then checks to see if the frame ticks is less than one thousand divided by the max fps. If it is then the `SDL_Delay()` function is called and the value of one thousand divided by the max fps minus the frame ticks is sent. Finally the fps is returned. The calculate fps function firstly sets up four static variables consisting of the number of samples, an array of frame times, the current frame and the previous ticks which is equal to the value of `SDL_GetTicks()`. The current tick is then calculated by calling `SDL_Ticks()` again and the frame time is then calculated by taking away the previous ticks from the current ticks. The modulo value of the current frame time and the number of samples is calculated and that position in the frame times array is set to the calculated frame time. The previous ticks are then set to the current ticks and the current frame is increased. A count variable is then created and if the current frame is less than the number of samples the count variable is set to the current frame, if not the count variable is set to the number of samples. The average frame time is then calculated by adding all of the used values in the frame time array together and then dividing them by the count variable. if the frame time average is above 0 the fps is set to one thousand divided by the frame time average. If not it is set to sixty.

Schedule

Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Finish main engine features	x	x														
Engine on Android		x	x													
Cell generation / death		x	x													
Cell movement				x												
blood vessels				x	x											
Score system					x	x										
Mutations								x	x							
Menus									x	x						
Cell Energy											x	x				
1 hallmark of cancer												x	x			
Bug Fixing														x	x	x