# TECHNICAL OVERVIEW

Object Factory

Components

Game Object Compositions

Other Systems and Managers

CORE ENGINE

Major Systems:

* Core (core.cpp)
* Factory (factory.cpp)
* Physics (physics.cpp)
* Graphics (graphics.cpp)
* Audio (audio.cpp)
* Game Logic (logic.cpp)

Core Components

* Transform
* Sprite
* Rigid Body
* Editable
* Sound

Emitter

Major Technologies

* OpenGL 3.3
* Custom 2D Physics
* ImGui
* FMOD Studio

# CODING METHODS

In order to comply with Visual Studio conventions and maintain a somewhat self-documenting structure, it is expected that every C++ class or object belongs to a group of similar objects. Each group is given its own folder, and each folder is given its own Visual Studio filter. For instance, the camera, sprites, and textures are all handled by some portion of the engine’s graphics. All of these files then are put into an “engineGraphics” folder, and all contents of this folder are put into an “engineGraphics” filter.

The only real file naming convention can be described as “some sort of camel case”. That is, either “camelCase.cpp” or “CamelCase.cpp” would be acceptable, and each file name should indicate one of the following

* The file’s primary purpose
* The primary C++ class found in the file

Nearly every .h (C/C++ header) file is given a macro-defined “code guard”:

|  |
| --- |
| #ifndef DEMO\_H  #define DEMO\_H  void demoFunction();  #endif |

All C/C++ files are expected to be in the folder that the Visual Studio project reads from (“GAM200\_PROJECT”). Other files and sub-projects are to be put into the same directory that this folder is in.

|  |
| --- |
|  |

A DigiPen git repository is used; all team members have been trained in using SourceTree in order make changes available to the rest of the team, and ensure that changes other team members have made are intact. When a merge fails, procedure depends on which file the merge failed for, and how simple the merge looks. (“Mine” refers to the working index/files of the merger, “they” refers to the last-known person to modify the files for which the conflict is failing).

* GAM200\_Project.vcxproj.filters
  + Resolve using “mine”, but add the files “they” added.
* \_EntryPoint.h
  + This file is very easy to modify, so resolution can be ignored.
* Simple C++ Implementation
  + Stage lines, but test compilation before pushing the merge.
* Highly complex (or non-documented) C++ Implementation
  + Contact whoever “they” happen to be
  + Failing that, or in cases where the merge causes the project “break” in a non-trivial manner, get Nolan.

# DEBUGGING TOOLS

Our team-built debugging tools mainly consist of:

* The standard command line window attached to a standard C++ project
* A system that allows us to create additional command line windows.
* A debug drawing system that is able to draw lines, circles, and rectangles.
  + In most cases, the function parameters are self documenting, with the exception that the “color” is ignored – all debug drawing is done in black.

|  |
| --- |
| Function prototype:  void debugDrawLine(Vector3 startPoint, Vector3 endPoint, Vector3 color); |
| Sample call:  debugDrawLine(  Vector3(mouseDownPos, 0),  Vector3(mouseUpPos, 0),  Vector3());//last parameter ignored |

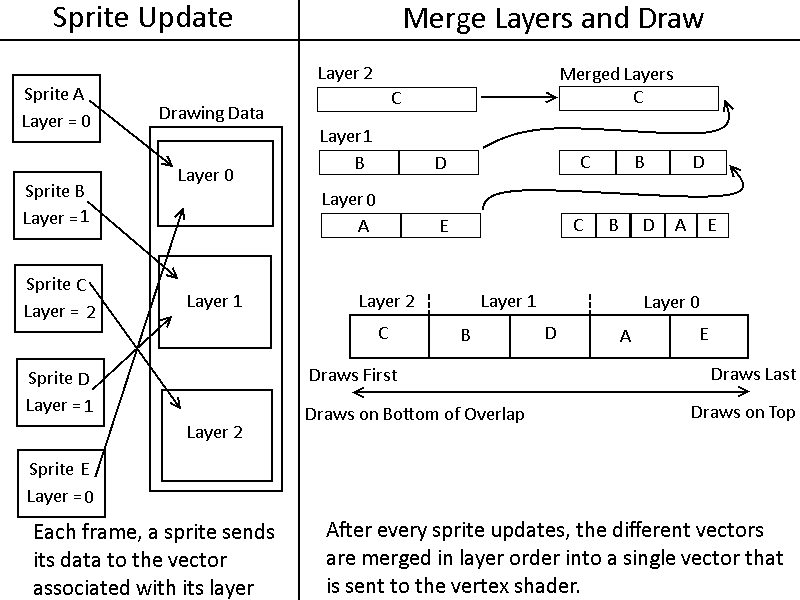
* + This call is used to draw a line constructed from a mouse click-and-drag.
* An assertion system that consists of warnings and errors.
  + Both AssertionWarning and AssertionError write to a text file and the debug console, but an AssertionError extends the standard C++ exception, and so stops the game (because nothing should catch it). AssertionError is only used when unexpected behavior occurs at a point where it could cause more unexpected behavior – for instance, attempting to delete an object that does not exist. This allows the origin of a bug to be quickly found.
* A frames-per-second meter.
* Sprites can be given text that refers to other variables in game.
* Debug tools are only on if the designer chooses to use them

# GRAPHICS OVERVIEW / ART PIPELINE

**Core Drawing System**

The Graphics Engine utilizes OpenGL 3.3 for accessing core graphics functionality, GLEW 1.1 for initializing OpenGL, GLFW 3.1 for window creation, and the latest stable release of SOIL (released on July 7, 2008) for loading textures.

Our graphics engine uses basic shaders. 2D sprites are drawn to the screen through a batching system, in which data from all sprites is collected each frame then sent to the graphics card in a single large batch.

The main game camera uses orthographic projection, and draw order is controlled through a sprite layer system that is integrated into the batching system. All sprites (and other data our engine renders) are found in an array of vectors – the vector that the data is found in determines the drawing layer the data will be drawn in.

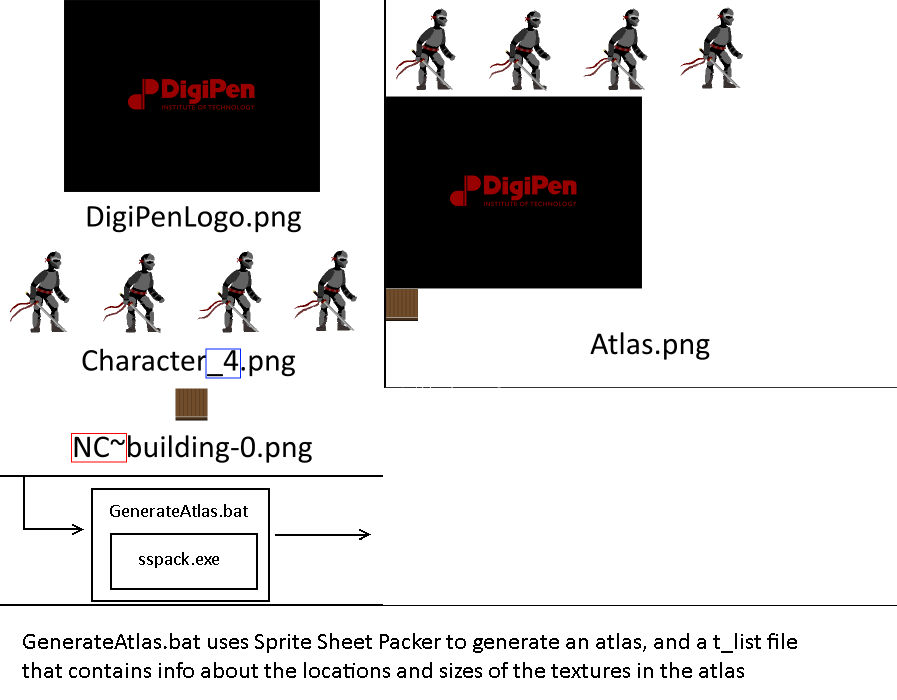
A batching system has the benefit of reducing the number of draw calls and consequently reducing resource usage. In addition, our batching system made it very easy to implement a sprite layering system that supports as many layers as designers need. For simplicity’s sake, the following example is presented with only three layers.

**Art Pipeline**

Instead of swapping textures in and out of the graphics card, all needed textures are packed into a single texture atlas as a pre-build step to running the game. However, this process is fairly automated. A third party tool called Sprite Sheet Packer does the majority of the work here, but the command line interface of Sprite Sheet Packer is wrapped in a batch file for convenience and time saving purposes.

To store certain data that Sprite Sheet Packer has no way of inferring, certain tags in filenames can be interpreted by the engine. For example, Character\_4.png has the tag “\_4”, which indicates this textures is a four frame animation. Another tag is the prefix NC~, which indicates that this sprite should not have collision if it is created as a tile.

An unfortunate side effect of using Sprite Sheet Packer, however, is that it expects to pack individual sprites, but not larger tilemaps. Designers/artists did not want to have to make tilemaps tile-by-tile, so we use a third party tool called ImageMagick to slice tile maps so that their individual tiles can be added to the atlas. Since ImageMagick is primarily a command line tool, a batch file in this case was less for convenience and more of a necessity.



//tList\_Atlas.png

//NOTE: Data is in form:

//X offset, Y offset, Width, Height

Character\_4 = 0 0 768 192

DigiPenLogo = 0 193 512 384

NC~building-0 = 0 578 64 64

Generating the atlas and tList file is a pre-build step. At runtime, the engine will read the tList file. Textures are stored in a map in the graphics manager. The key is a just a string, the name of the texture, and the value is the texture itself, as an AtlasTexture object. AtlasTexture objects are basically containers for storing the relevant information that was read from the tList file.

**Animation**

With the knowledge that Sprite Sheet Packer will never “bump” an animation to the next line, and the width and number of frames are made available as part of the packing process. This makes animation is trivial, as all information needed to animate a sprite is accessible. The animation algorithm just determines what the texture coordinates of the current frame are. There’s two steps in animation: determining whether or not to change the current frame, and updating texture coordinates based on the current frame. Determining whether or not to change the current frame is just based on a timer, calculating texture coordinates is done by calculating the four bounds of the shape that defines the texture.

|  |
| --- |
| //Core animation functions  GLfloat AtlasTexture::getBottomY()  {  return (offsetY + frameHeight) / (float)atlasHeight;  }  GLfloat AtlasTexture::getTopY()  {  return offsetY / (float)atlasHeight;  }  GLfloat AtlasTexture::getLeftX()  {  return ((currentFrame \* frameWidth) + offsetX) / ((float)atlasWidth);  }  GLfloat AtlasTexture::getRightX()  {  return ((currentFrame \* frameWidth) + offsetX + frameWidth) / ((float)atlasWidth);  } |

**Sprite Text**

Although this feature was not fully fleshed out due to being low priority, our engine does support limited sprite text rendering. At this point, its best and most reasonable application is in displaying debug sprite text in the game world. This can be accomplished with a simple call to renderText, which will draw sprite text to the frame in which it was called.

|  |
| --- |
| static void renderText(  std::string message, Vector3 position, Vector3 scale); |

# PHYSICS OVERVIEW

The physics engine and libraries were originally written to accommodate C code, and has internally undergone several revisions.

Our rigid body supports the following properties and functionalities:

* Position
* Velocity
* Mass
* Acceleration
* Friction
* Restitution
* Ghost/Non-ghost
* Static/Dynamic
* Self-serialization
* Binding to the Zilch scripting language

Colliders in our engine don’t rely on our rigid body definitions. Box-to-box and box-to-circle collision detection are both supported. Colliders are self-serializing, and are bound to the Zilch scripting language.

The physics engine also comes with

* A custom library for managing 2D and 3D vectors
* A manager that iterates through each body to simultaneously handle gravity, velocity, and collision resolution
* A system for preventing objects from penetrating each other while the collision is being resolved
* A ray-casting system – a ray-cast can hold information regarding every object it collides with.
* A partial binary map system, allowing for a level’s tile map to be without a true rigid body.
* Support for triggers – when a collision occurs, a trigger can be set-up to automatically run in direct response to the collision. In technical terms, this is a function callback.

Euler integration is used across the entire Physics system.

In order to support gameplay, the player state is specially handled, allowing the player controller to know whether the player is grounded or not.

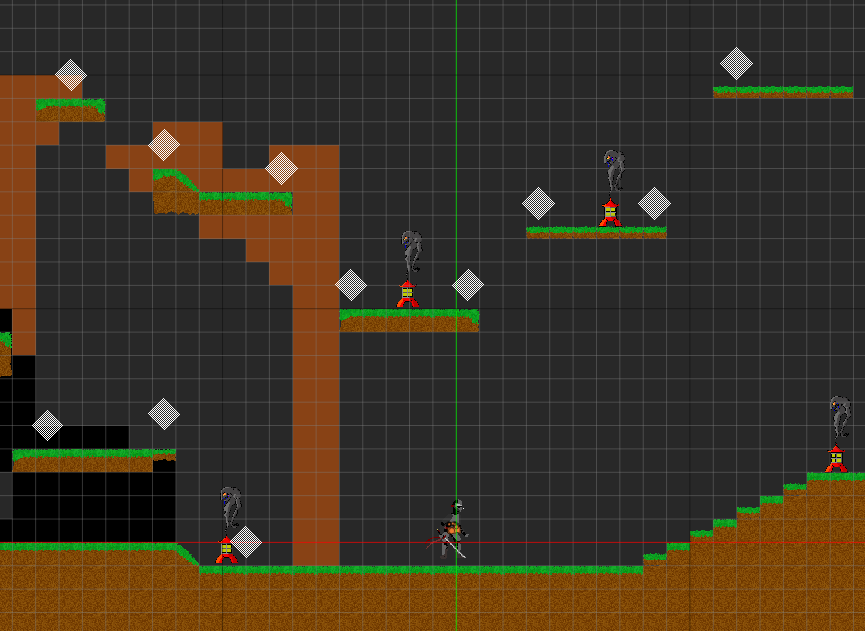
# BEHAVIORS OVERVIEW

Primary Enemy AI Structure:

* Our AI uses very simplistic state controls that allow enemies to alternate between states such as attacking, chasing, and pacing.

I:\GAM 200\Art Assets\AI_state_machine.png

* Our AI uses distance computations as well as ray-casting to determine if the player is in range to chase or attack.
* Upon moving close enough to the player, the enemy will perform an attack that sends an event to the player to determine whether or not damage is dealt and exactly how much damage is dealt.
* We also have multiple enemy AI algorithms prototyped for A-star pathing, point pathing, and platform navigation logic that will be featured on more difficult enemies as they are ready to be moved in-engine.



# EDITOR OVERVIEW

**Editor Overview**

Our game uses an in-engine editor, built primarily using ImGui. An in-engine editor was chosen so that we could easily use existing systems, mainly graphics. A standalone editor would require its own art pipeline and rendering system in addition to the one already required by the game. An in-engine editor significantly reduced the amount of work required to build the editor and reduced possible errors by having two versions of a rendering pipeline.

The editor has three main modules, Level Tools, Tilemap Tools, and Entity Tools.

**Level Tools**

The editor is in-engine, and interfaced with through ImGui. This allows designers to quickly experience how their level will perform. A level consists of (and is saved as) two files. The first tile contains the level’s name and tilemap. The second file contains all information regarding objects and their components. In other words, it contains serialized game object data that the factory can use to recreate objects as the designer has defined them.

**Tilemap Tools**

The editor also includes a tilemap editor that displays all textures available in the game and allows the user to change the appearance of any tile to be that of any available texture. Tiles only consist of a sprite and collision information (if applicable).

**Entity Editor**

The entity editor allows for dynamic translation (click and drag), rotation (ctrl-click and drag), and scale (shift-click and drag). While in editor-mode, entities respond to mouse events and their properties are therefore modifiable as with every other property.

Editor allows for the addition and removal of components. Because many of these components are bound through the Zilch scripting language, their functionality is not shown until the designer reloads the level (which requires a single click).

Some component properties can be tweaked at runtime, provided that the programmer has correctly bound the component in question to the editor.

For most components, generalized logic is used, but the identity and functionality of the property we’re reading from or writing to is not necessarily known.

# AUDIO OVVERVIEW

Audio integration relies on FMOD Studio 1.06.08 and the FMOD API, meaning that FMOD Studio .bank files are used for background music and sound effect files.

The game engine’s sound manager is designer to wrap around FMOD studio’s API – more specifically, we create a container using Fmod::Studio::EventInstance to loop through all sound files. Sounds can be created, updated, released, and unloaded. Throughout this process, the FMOD API-supplied flags are used to ensure that the audio files that we attempt to load are actually loaded correctly.

A sound emitter class was also written to help facilitate this process, wrapping around FMOD event instances.

This wrapper also allows us to create FMOD event instances of each file in the .bank file. Additionally, through our wrapper, it is possible to pause audio by setting the flag on a sound emitter.

Adding music without recompile the game engine is possible through the following process:

* drag the new music file into FMOD studio, generating a new bank file
* copy it to the game engine’s assets, overwriting the old bank file
* The bank allows the engine to access sound files found in the bank file