CS529 Final Project – Asteroids

Francis Joseph Seriña
60001514, francis.serina@digipen.edu

December 6, 2014

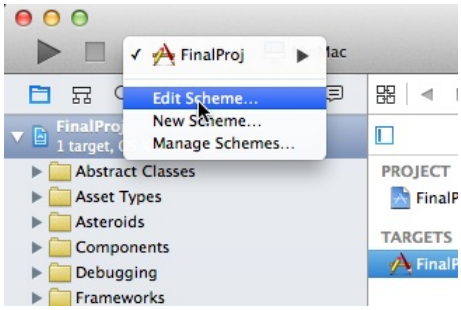
Building the Game

Windows

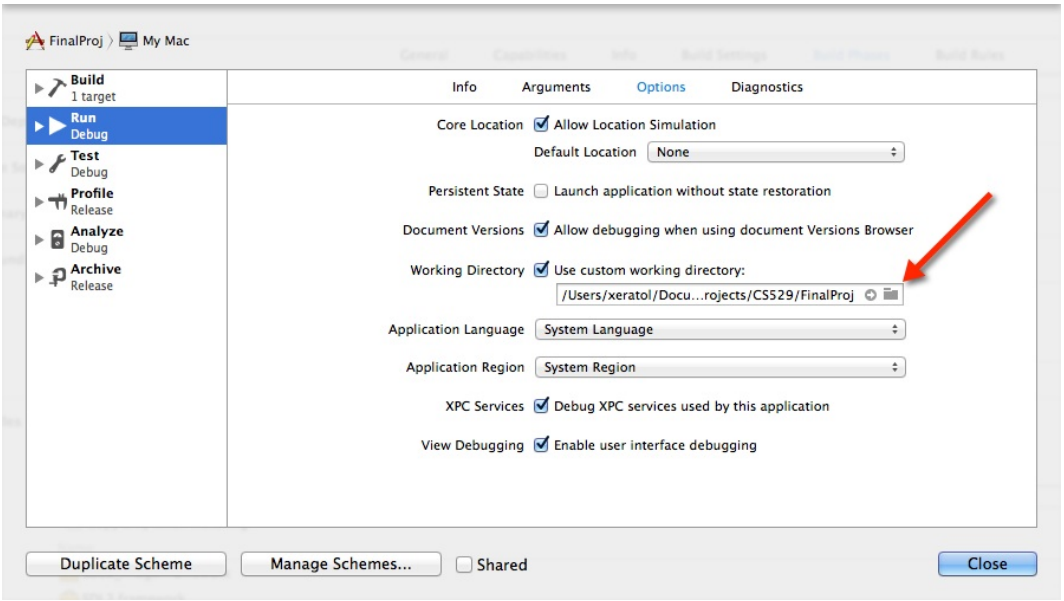
Navigate to the `FinalProj/visualstudio` folder and open `FinalProj.sln` or `FinalProj.vcxproj` using Visual Studio 2012 or higher. Press Ctrl + F5 to build and run the project.

Mac OSX

Navigate to the `FinalProj/xcode` folder and open `FinalProj.xcodeproj` in XCode. Set the current working directory to the `FinalProj` folder.



Edit Scheme



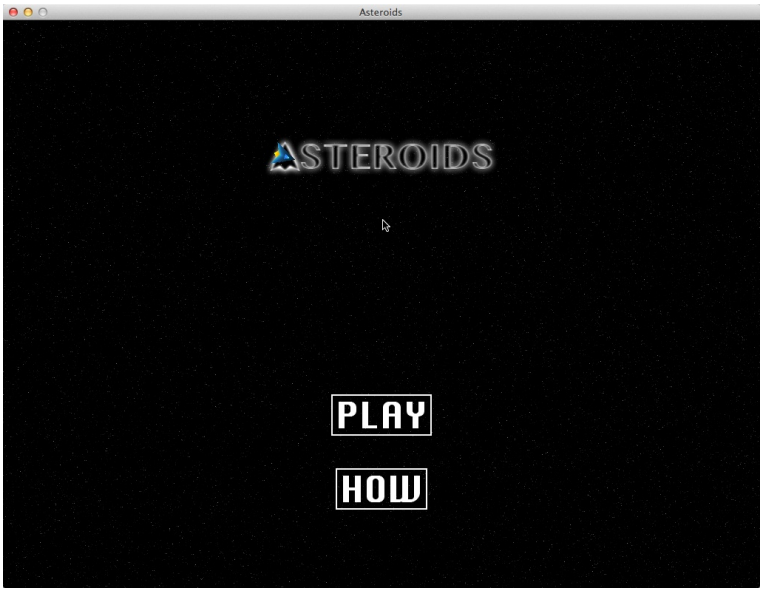
Edit Working Directory

Press Close. Select Run from the Product menu or hit CMD+R to build and run the game.

Asteroids Game

Main Menu

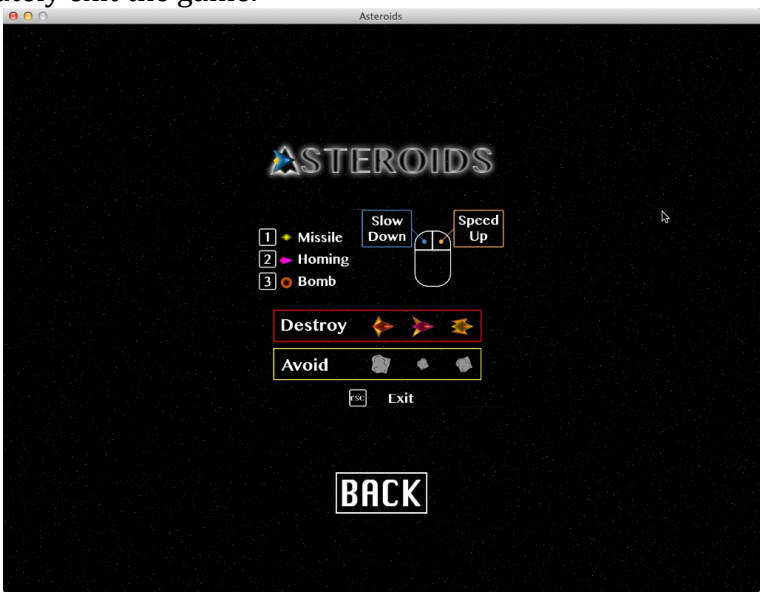
Upon successfully compiling and running the game, you will see the Main Menu. There are 2 self-explanatory buttons here. Notice that the player's ship is always facing the mouse.



Main Menu

Player Controls

Clicking the *How* button leads you to the controls screen. This screen is quite intuitive. Notice that you can move your ship by right clicking within the screen even without pressing the *Play* button. Pressing *Esc* anytime will immediately exit the game.

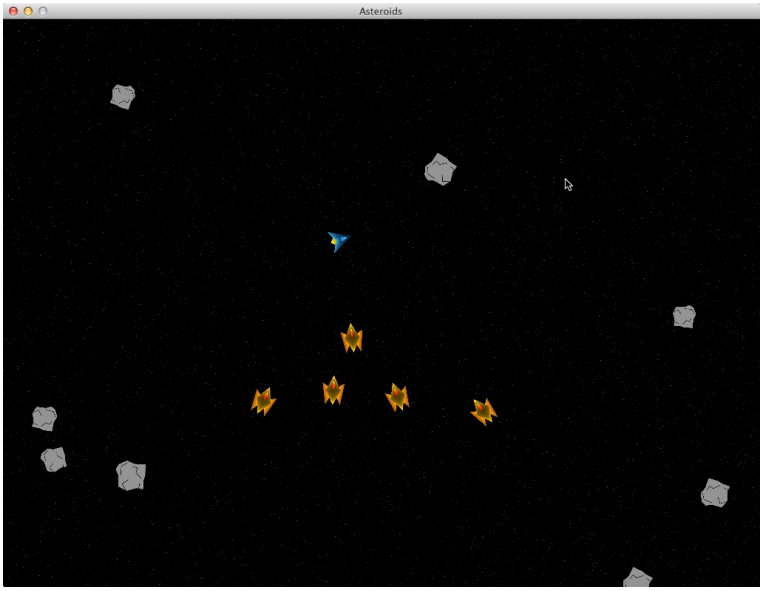


Controls Screen

Weapons

The player has 3 weapons: Missile, Homing, and Bomb. The Missile is a projectile that moves forward and detonates after some time. The Homing is a projectile that looks for and tracks down the nearest Enemy. The Bomb is a weapon that damages all enemies within the explosion. To use a weapon, press the corresponding key on the keyboard.

Enemies
There are 3 types of enemies. The simplest one moves forward until it goes beyond the screen. The second goes to a series of waypoints on the screen. The third one tries to chase and collide with the player.



Last level with chasing enemies

The enemies can be destroyed by hitting it with any weapon or by leading it to an asteroid.

Obstacles
There is only one obstacle, which is the Asteroid. The asteroids are indestructible. Hitting them with a weapon simply bounces them off. Colliding with an asteroid destroys you. However, enemies are destroyed when they collide with the asteroids as well.

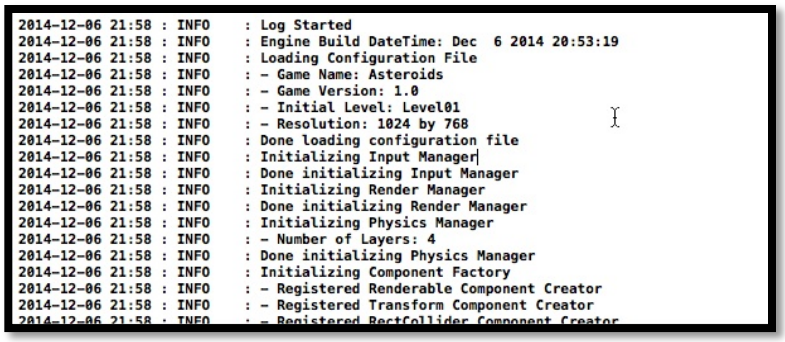
Player Objectives
Destroy all enemies in the 3 waves to win.

Losing Conditions
Lose your ship.

Debugging
In the course of developing the project, it is critical to see the inner workings of the engine. Finding memory leaks, testing collision algorithms, visually displaying physics, etc. are critical to keeping the engine running smoothly.

Console/Log
The first debugging assistant created was a Logging System. The logging system must be completely independent from all other systems in order to log any events happening in the game. By default, the logging system outputs to both the console and a log file. The default log file is `logs.txt`. See `SLogger.h/cpp` for more information such as disabling console output and setting the log file.

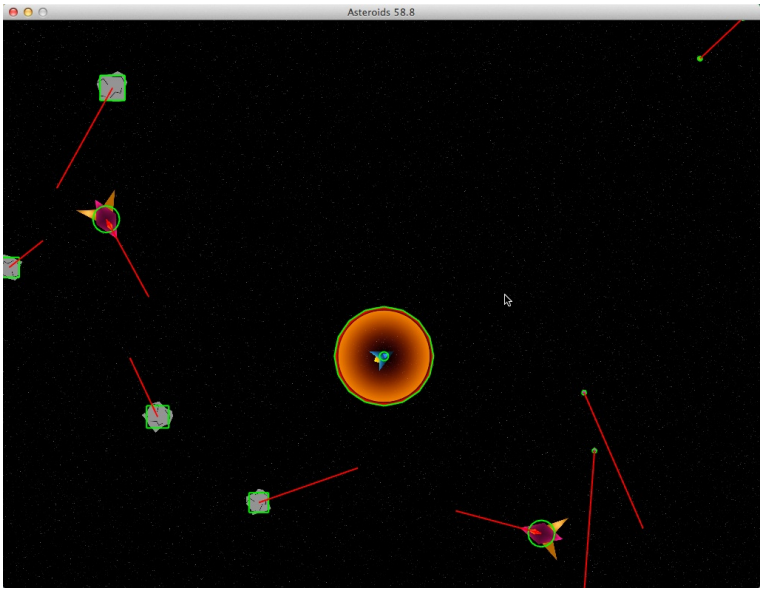
There are 3 methods in logging events in the game: Information Log (ILOG), Warning Log (WLOG), and Error Log (ELOG). The only difference between them is the prefix (`INFO, WARNING, ERROR`) for the event. The 3 methods are used similar to `printf()`.



Sample Log

Debug Drawing

Debug Drawing is used to display the colliders and velocities. Debug Drawing is enabled by calling `sGame->SetDebug(true)`. For the Asteroids game, pressing 'D' on the keyboard enables debug drawing. Colliders are drawn in green while velocities are drawn in red.



Debug Drawing

Frame Rate

The frame rate is on the title bar when the game has debug drawing enabled.

Object Architecture

The heart of the game engine lies on its object architecture.

GameObject-Component Relationship

Game Objects are the objects in the game which either have logic, is visible, or both. Game Objects can be further decomposed into more modular parts called Components. In this game engine, a Game Object is a container of various components. A Game Object can have as many components as it needs. Components belong to exactly one game object, which is referred to as the *owner*. The ownership is *weak* and will be shown later. A Game Object can only have at most 1 of the built-in components. However, custom components can be added multiple times. See `AGameObject.h/cpp` and `AComponent.h/cpp` for more information.

Game Object Pool

All Game Objects register themselves into the Game Object Pool upon creation. A Game Object should be destroyed by its public method `Destroy()` which does its own clean up – queueing itself for destruction, informing all its components to destroy themselves. The pool does the actual deletion.

Components

Components are small units of the Game Object that perform different tasks. The following are built-in to the game engine: **Transform** (`CTransform.h/cpp`), **Renderable** (`CRenderable.h/cpp`), **Updateable** (`CUpdateable.h/cpp`), **RectCollider** (`CRectCollider.h/cpp`), **CircCollider** (`CCircCollider.h/cpp`), and **Rigidbody** (`CRigidbody.h/cpp`).

Transform does not perform logic or display anything but it contains relevant data that other components need. The data it contains are Position, Rotation, and Scale.

Renderables are handle displaying meshes and textures on screen. Renderable needs the Transform component.

Future Work: Camera Component

Updateables by itself does nothing but this is an avenue for creating custom logic depending on the need of the game.

RectCollider and CircCollider are used to determine collisions between game objects. Colliders need the Transform component.

Rigidbody are used to perform physics simulations involving mass, velocity, resultant force, and collision reaction. Rigidbody needs the Transform component.

Component Managers

All components, except the transform, are managed by a specific Manager and are stored in its respective component pool.

Renderables are handled by the **Render Manager** (`SRenderManager.h/cpp`). Updateables are handled by the **Game Manager** (`CGameManager.h/cpp`). RectColliders, CircColliders, and Rigidbodies are all handled by the **Physics Manager** (`SPhysicsManager.h/cpp`). Transforms are managed by the Game Object that owns it.

Component Pools

All components, except the transform, are *strongly* owned by their respective pools. Renderables are all contained inside the Pool inside Render Manager. Updateables are all contained inside the UpdateableObjectPool. RectColliders and CircColliders are all contained inside the ColliderObjectPool. Rigidbodies are all contained inside the RigidbodyObjectPool.

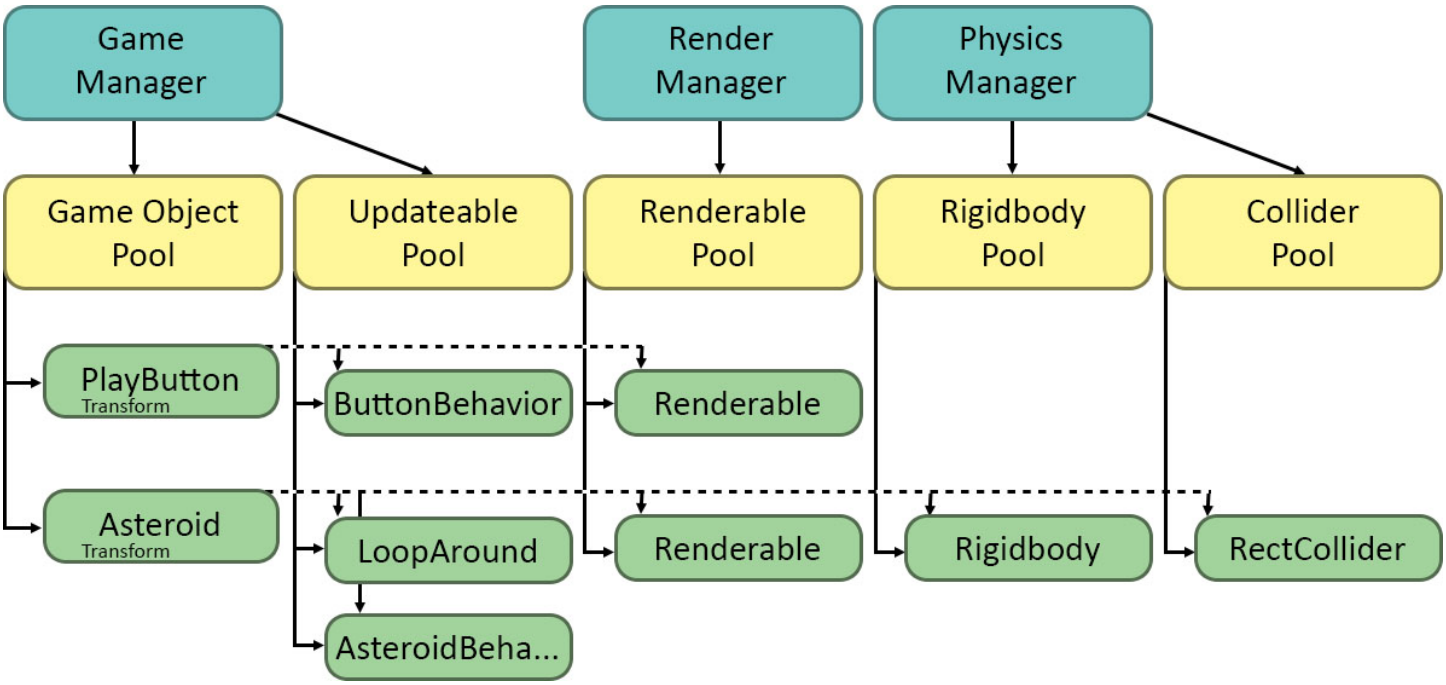
Future Work: It would be wiser to have the pools incorporated into the managers. Right now, only the Renderables/RenderManager does this. The rest have a pool completely separate from their managers. Also, each pool sets IDs. It is possible to have a Renderable and Rigidbody component with both IDs equal to 1. It would be better if there was only one component ID counter.

When a component is created, the first thing it does is attach itself to its respective pool.

To destroy components, it should be done from the owning Game Object via the `RemoveComponent()` method. This way, the Game Object is aware of the components removal. The component will automatically inform its corresponding pool to destroy this component.

Game Object Manager

The Game Manager does a lot of things; one of them is keeping track of all the game objects. Technically, all game objects are *weakly* owned by the Game Manager. The Game Manager also controls the other managers.



Manager-Pool Relationship

In the diagram above, the Game Manager has direct access to the Game Object and Updateable Pools. The Render Manager has direct access to the Renderable Pool. The Physics Manager has direct access to the Rigidbody and Collider Pools. A Game Object called `PlayButton` has a Transform, which the game object manages, and has 2 components: ButtonBehavior (custom Updateable) and a Renderable. The PlayButton weakly owns these 2. Instead, their respective pools have the strong ownership over them. The `Asteroid` game object is more complex containing 5 components where 2 of them are both Updateables.

Communication

The communication system is a way for Components to receive events from the Game Object. Note that a component need not be owned by the game object it wishes to listen to.

Invoker-Listener Relationship

All Game Objects are invokers. Meaning, the game object queues messages to be processed by its listeners. See `ProcessAllMessages()` from `CGameManager.cpp` for implementation. To invoke a message, simply add the message to the message queue via the `AddMessage()` methods in `AGameObject.h`.

All components can become listeners via the `AddMsgHandler()` or `Connect()` method of `AGameobject.h`. The listening component should have a void method that accepts a const pointer to an AMessage.

If data needs to be passed, rather than just invoking an event, one must first make a custom message, inheriting from AMessage. See `TargetSetMessage.h` or `NewEnemyMessage.h` for examples. The data is filled in before calling `AddMessage()` on the Game Object.

Let's have an example. Consider the following snippets from `SetTarget.cpp` and `TurnToTarget.cpp`. These are both custom Updateables used by the Homing weapon and the chasing enemy.

```
void SetTarget::Start() {
    GetOwner()->Connect("OnTargetLost", SetTarget::OnTargetLost);
    OnTargetLost(0);
}

void SetTarget::OnTargetLost(const GD4N::AMessage *msg) {
    GD4N::AGameObject* target = 0;
    // find a new target
    if ( target ) GetOwner()->AddMessage(new TargetSetMessage(target));
}
```

Snippet from SetTarget.cpp

```
void TurnToTarget::Start() {
    GetOwner()->Connect("OnTargetSet", TurnToTarget::OnTargetSet);
}

void TurnToTarget::OnTargetDestroy(GD4N::AMessage const * msg) {
    _target = 0;
    GetOwner()->AddMessage("OnTargetLost");
}

void TurnToTarget::OnTargetSet(const GD4N::AMessage *msg) {
    GD4N::AGameObject * target = dynamic_cast<TargetSetMessage const *>(msg)->GetTarget();
    _target = &target->Transform()->Position();
    target->AddMsgHandler("OnDestroy", this, &TurnToTarget::OnTargetDestroy);
}
```

Snippet from TurnToTarget.cpp

`Start()` is executed in the beginning of the frame after the Updateable was created.

SetTarget connects the SetTarget::OnTargetLost method to listen to *OnTargetLost* events from the owning game object. It also calls the method to search for an initial target. When a target is found, it then queues a TargetSetMessage containing the target.

TurnToTarget connects its TurnToTarget::OnTargetSet method to listen to *OnTargetSet* events sent by the owning game object. When an *OnTargetSet* message is invoked, it gets the target from the message, points to its position and registers an *OnDestroy* event listener on the target. When the target is destroyed, TurnToTarget will be made aware via the OnTargetDestroy method. This method will remove the target (to prevent access violations) and add an *OnTargetLost* message in the message queue. This will then be caught by SetTarget and find a new target.

Future Work: The Message Handler nodes are stored using std::list. Removal of specific listeners, because the component is to be destroyed, has a linear running time instead of constant. Solution is to create my own doubly linked list.

Data Driven Design

To make the game engine as versatile as possible, all game objects and components should be created from files. There will always be a few hard-coded data such as the initial configuration file which is set when calling `Run()`. See `main.cpp` for an example.

Config File

The config file will point to all other files. The config file has a <key,value> format that are white-space delimited. As such, keys and values are not allowed to have whitespaces. The following is a list of accepted key-value pairs for the config file. Keys need not be ordered as shown.

Key	Value Format	Notes
GameName	Any string	Displayed in the title bar
GameVersion	Any string	For updating purposes
Resolution	2 Uints	Width and Height of the game
AssetList	Path to file	Multiple AssetLists can be specified as long as asset IDs are unique
PhysicsConfig	Path to file	Only 1 Physics Config file should be specified
ArchetypeList	Path to file	Multiple ArchetypeLists can be specified as long as names are unique
LevelFile	Path to file	Multiple LevelFiles are allowed as long as level names are unique
InitialLevel	Any string	Only 1 InitialLevel should be set. Should exist in one of the LevelFiles

Only the ArchetypeList is optional. All other keys must be set.

AssetList

An asset list contains the list of assets to be loaded. All entries should follow the following format:

```
<Asset Type> <Unique Asset ID> <Path to Asset>
```

As of the moment, only `Mesh` and `Texture` are the allowed Asset Types. Asset IDs are just numbers. Assets are loaded via the AssetFactory. See `AssetFactory.h/cpp` for implementation.

Future Work: Add material, sound effects, background music, and font asset types.

ArchetypeList

An archetype list contains the list of archetypes to be loaded. All entries should follow the following format:

```
<Unique Archetype Name> <Path to Archetype File>
```

The Archetype Name is used by the Level Files. Also, it is now allowed to use *GameObject* as an Archetype Name. The archetypes are loaded into memory for faster creation. See `LoadArchetypes()` from `CGameManager.cpp` for implementation.

Archetype File

An archetype is a pre-made game object. Archetype files contain a list of components and their default values. The following is the `MissileArchetype.txt`.

Missile Archetype

This archetype has 7 components: `Transform`, `Renderable`, `CircCollider`, `Rigidbody`, `DestroyOnCollision`, `DestroyAfterTime`, and `MoveForward2D`. Specific properties are set for all the components except Rigidbody and DestroyOnCollision. The properties are also using the key-value format. Though some keys require more than 1 value. For example; Position, Rotation, and Scale of the Transform component expects 3 real numbers. A semi-colon denotes the end of a component.

LevelFile

The LevelFiles contain a list of game objects and/or archetypes. Each game object should have the following format:

```
<Archetype Name or GameObject> <Game Object Name>
      <Component List and Properties>
;
```

The Archetype Name should exist in one of the ArchetypeLists. If the game object does not come from a template, simply use *GameObject*. The Game Object Name could be any string and does not necessarily have to be unique for the game. The Component List and Properties are the same for the Archetype File. A semi-colon is required to denote the end of a game object. See `level01.txt` for an example.

PhysicsConfig

The PhysicsConfig file would be discussed in the Physics section

Future Work: Create a GraphicsConfig which contains settings for the Graphics such as vsync, screen resolution, full screen/window mode, etc.

Component Factory

Since each game needs custom Updateables, to incorporate these custom behaviors into the serialization process, all custom Updateables should register itself to the Component Factory. See `AsteroidGame.cpp` to see how custom updateables add themselves to the Component Factory. See `ComponentFactory.cpp` for more information.

Graphics

The engine uses OpenGL 2.1 to draw. See `Init()` and `InitOpenGL()` in `SRenderManager.h/cpp` for more information.

Future Work: Use OpenGL 4.x using Vertex Buffer Objects, add programmable pipeline (shader) support.

Renderable Component

The Renderable Component uses a Mesh and Texture Asset to draw things on screen. The Mesh contains the vertices and the Texture contains the image to be drawn on the mesh.

Future Work: Use TRS Matrix instead of glTranslate, glRotate, and glScale. Create Material component to allow sprite sheet and animations.

Sprite Z Sorting

When a new Renderable is added to the pool, a flag, *_hasNewElements*, is set to true. See `CleanUp()` of `IComponentPool.h`. This flag is checked right before rendering all the objects. See `RenderAll()` of `SRenderManager.cpp`. If new elements are present, the Renderable components are sorted from least z to highest z. See `SpriteZSorting()` of `SRenderManager.cpp`. This allows for correctly rendering transparent textures.

Future Work: Camera Component

Physics

The Physics Manager does 2 things every frame: check for collisions and simulate physics through the Rigidbody. See `UpdateAll()` of `SPhysicsManager.cpp`.

Circle and Rectangle Collider

The engine has 2 built-in collider shapes – Circle and Rectangle. Note that these need the Transform component since the dimensions are multiplied by the Scale of the Transform. See `CollisionTest.h/cpp` for implementations.

Basic Collision Response

Collision Response is handled as an event/message by the Rigidbody component. See `ReactToCollision()` in `Rigidbody.h/cpp`.

Future Work: Compute for contact points for Collision Message. Fix penetration issues. Fix collision normals. Merge CollisionTest and MathUtils. Brute-force collision testing. Optimize using Binary Space Partitioning or Quad Trees.

PhysicsConfig

The Physics Config file contains settings for the Physics Manager. The following keys are accepted:

Key	Value Format	Notes
NumLayers	Uint	Number of Layers
Layer	Any string	The different Layer Names
LayerCollision	2 strings	2 Layer names that allow collision
Gravity	Real number	Not used

```
NumLayers    4
Layer        Enemy
Layer        Player
Layer        EnemyWeapon
Layer        PlayerWeapon
LayerCollision  Enemy Player
LayerCollision  Enemy PlayerWeapon
LayerCollision  Player EnemyWeapon
Gravity      0
```

Sample PhysicsConfig

Future Work: Enable global gravity

Layer-Collision Matrix

The Layer-Collision Matrix is a 2D bool array indicating which layer can collide with whom. Layers are set in the RectCollider or CircCollider components.

The sample physics config above would generate the following layer-collision matrix:

	Enemy	Player	EnemyWeapon	PlayerWeapon
Enemy	✗	✓	✗	✓
Player	✓	✗	✓	✗
EnemyWeapon	✗	✓	✗	✗
PlayerWeapon	✓	✗	✗	✗

Layer-Collision Matrix

If a collider has no specified layer, it means it can collide with everything.