GAMES ARCHITECTURE

HENRY PAUL

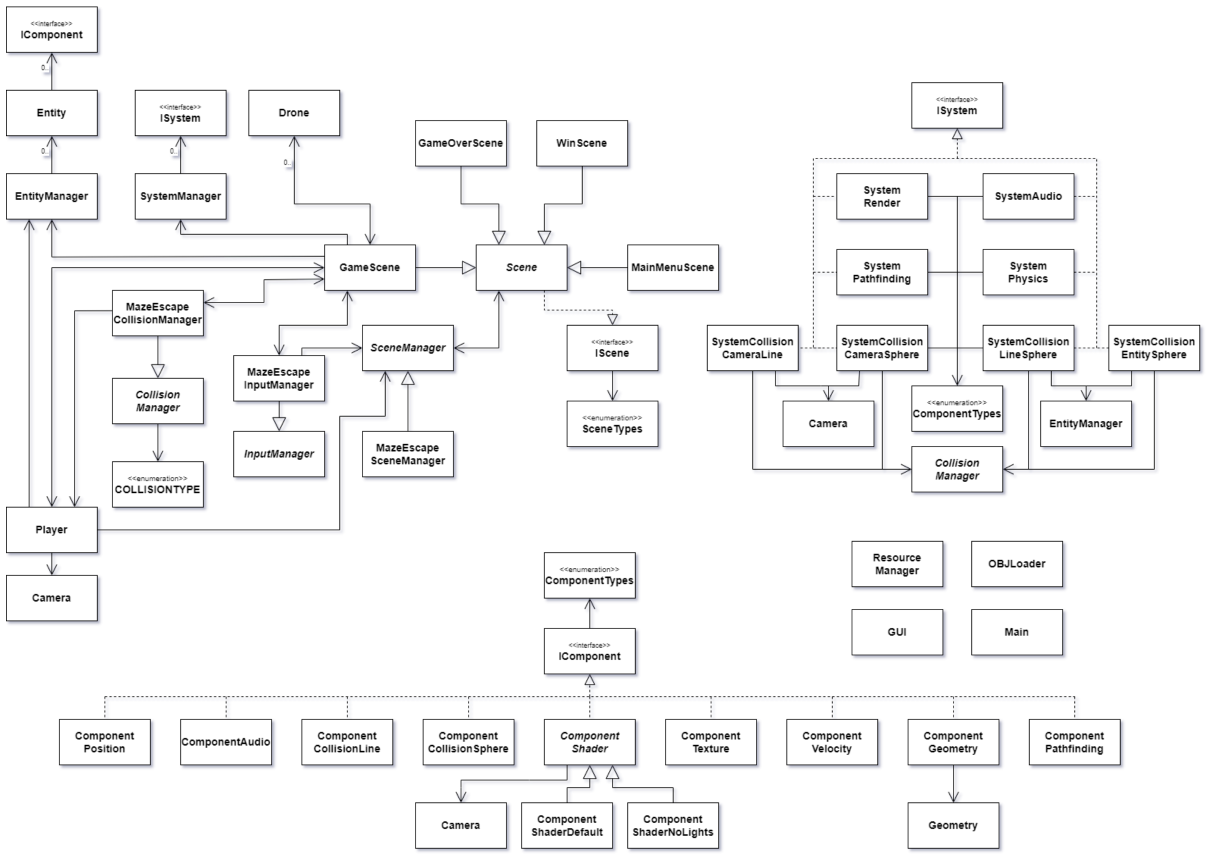
Contents

Games Architecture

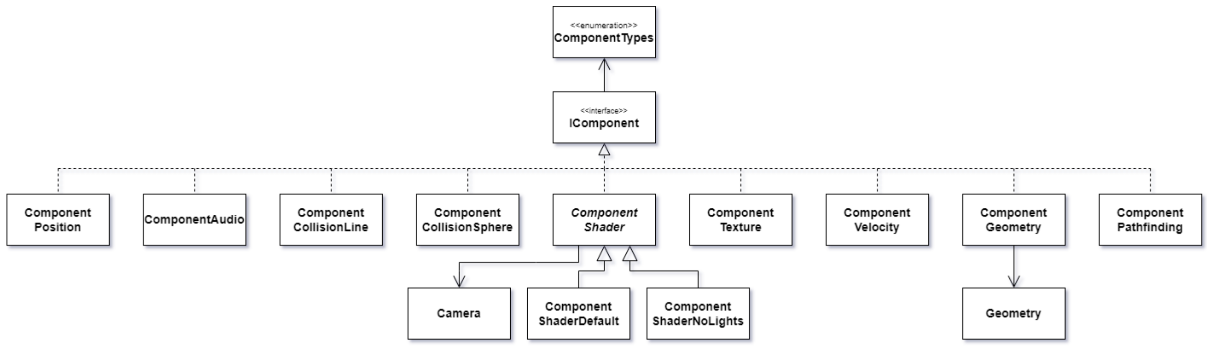
Chapter 1: Design and Critique

UML Class Diagrams

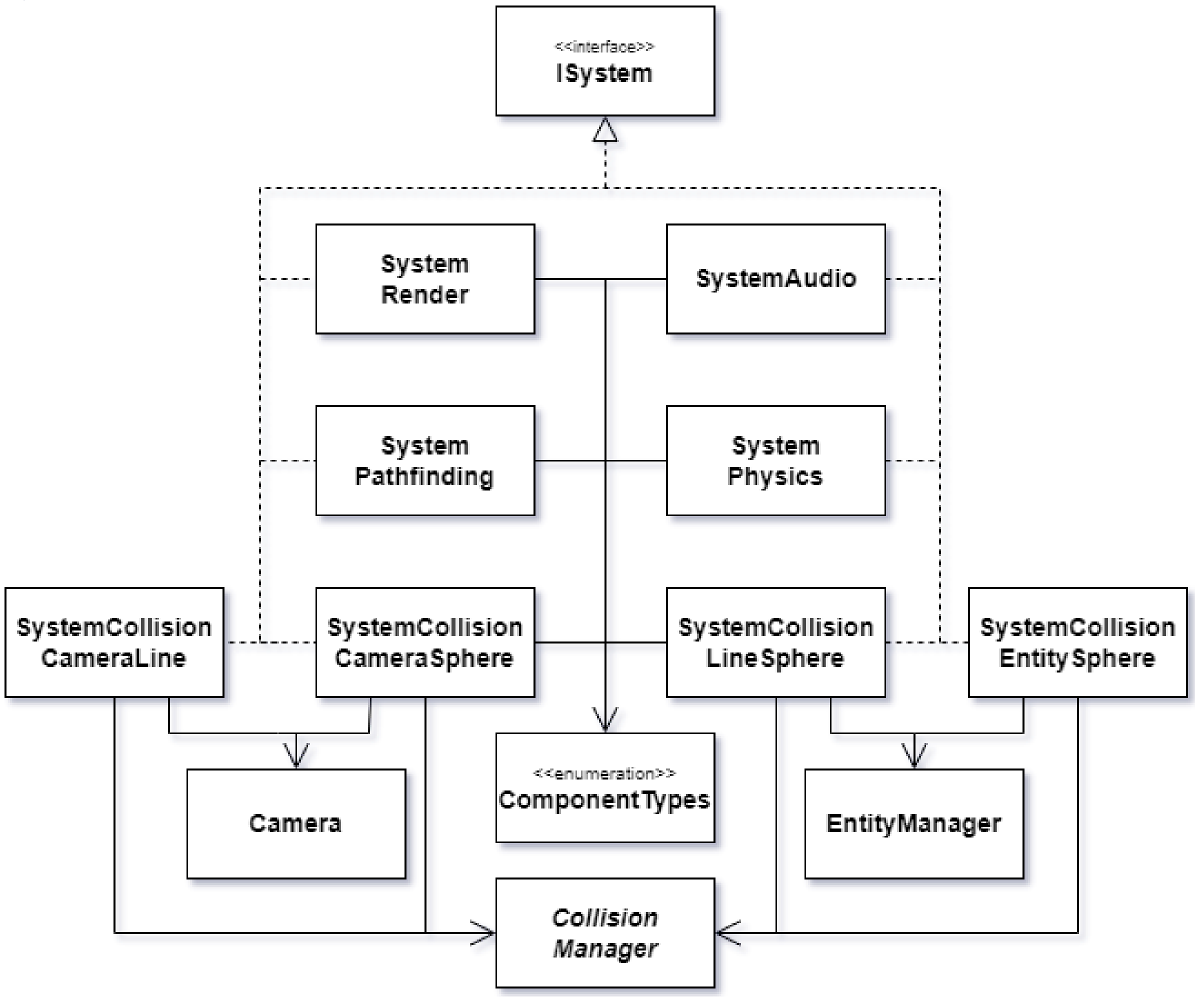
Overview



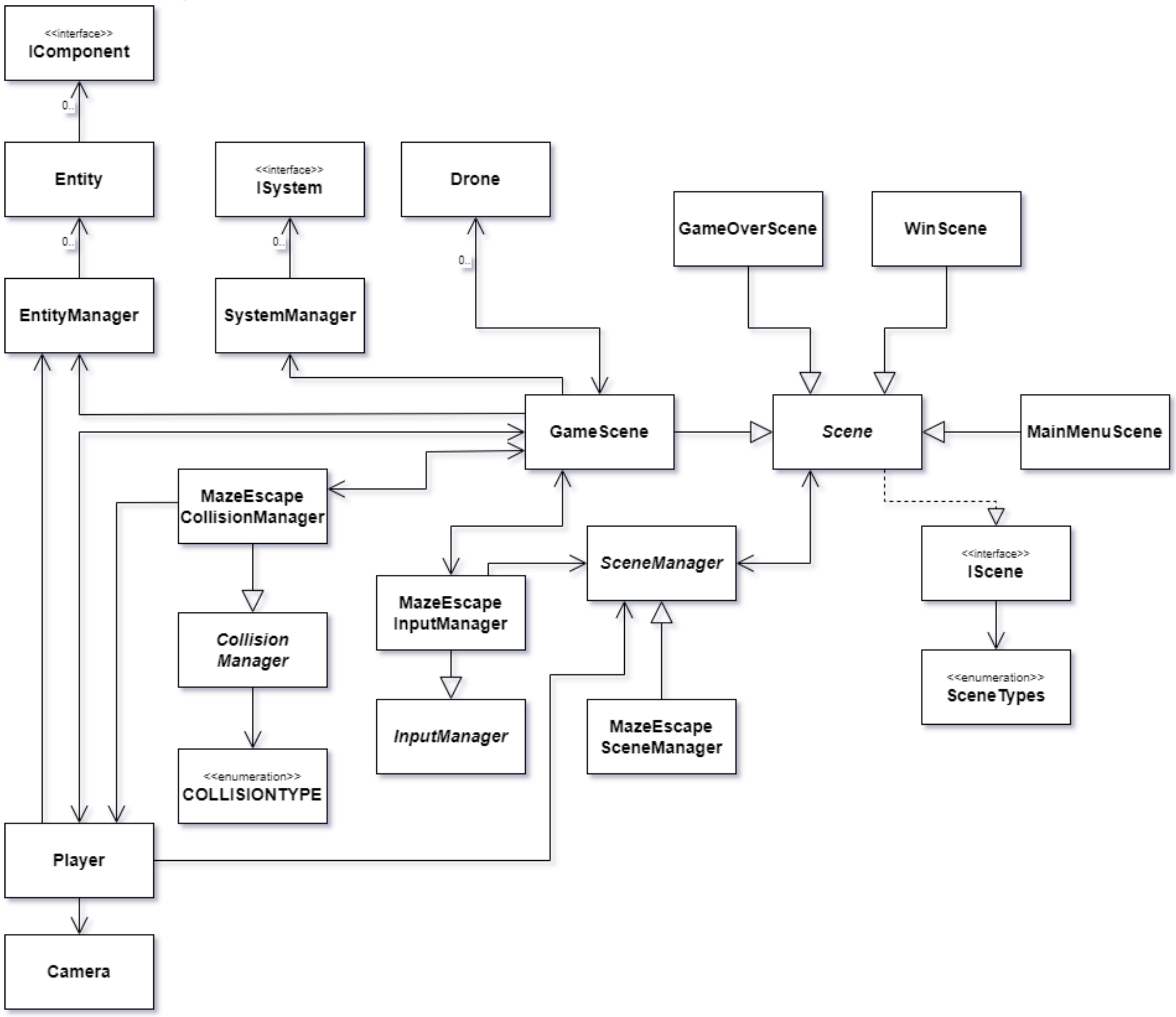
Component Overview



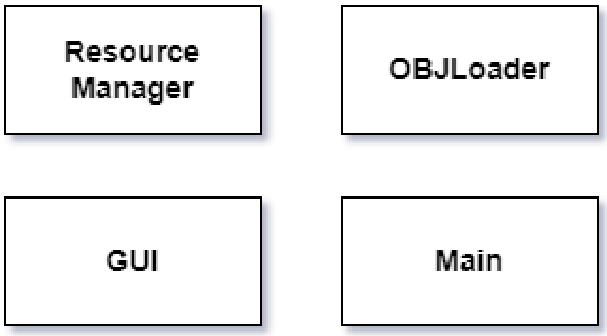
System Overview



Scene / Manager Overview



Static classes



Class descriptions – Engine Components

IComponent

This is an interface class in which all other component classes derive from. It ensures that every component has a way of retrieving its component type with a get function, in addition to a “Close” function. It is also where the ComponentTypes enum is located.

ComponentAudio

The audio component takes a file path as a parameter, as well as two boolean values that determine whether or not the audio should be looping and if the audio should play on creation. It uses the OpenTK audio library and the static ResourceManager to store and play the audio file.

ComponentCollisionLine

This component holds two Vector3 types, representing the start point of the line and the end point. This can be used to make an entity collidable, which will be detected by one of the later mentioned collision systems.

ComponentCollisionSphere

Same as the line collision component, except this one holds a float value representing the collision radius.

ComponentGeometry

This component takes a file path as a parameter and loads the model at that path using the static ResourceManager. The resulting is stored in the component as a Geometry type.

ComponentPathfinding

Original planning was to implement a true pathfinding system using the A* algorithm. However, due to time constraints, a much more simplified version of “pathfinding” had to be implemented. Instead of calculating a path, all paths are pre-defined and stored in this component as a list of Vector3 objects. The component also holds a boolean value, determining whether or not this component should be active, allowing a developer to toggle the movement in their game code. It also stores a float value representing the movement speed of the entity.

ComponentPosition

The position component holds two Vector3 objects, “lastPosition” and “position”. The last position vector allows the developer to implement a collision response by moving an entity back to its previous position in the event of a collision, if they wish.

ComponentShader

This is an abstract class that allows the creation of multiple different shaders that can be applied on a “per-entity” basis. It takes two strings for the filenames of the vertex shader and the fragment shader so they can be loaded by the static ResourceManager.

ComponentShaderDefault / ComponentShaderNoLights

These components derive from the previously mentioned shader component class. The default shader passes the default filenames to the base shader class so they can be loaded. It also has its own implementation of the abstract method “ApplyShader”. The reason this is abstract is because each shader can have different requirements in terms of what needs to be passed into them such as uniform values. Giving each shader component its own implementation of this method allows for those specifics to be set. The no lights shader component uses the same vertex shader file as the previous component but uses a different fragment shader. The default shader uses 4 lights in the scene whereas this one has no lights included in the fragment shader, meaning any entities with this component applied will not have any lighting calculations applied to them.

ComponentTexture

This component takes a file name string as a parameter and uses the static ResourceManager to store the result as an integer, which can be retrieved using the components get function.

ComponentVelocity

This component holds a Vector3 representing the velocity of the entity it is applied to, as well as the appropriate get and set functions.

Class Descriptions – Engine Managers

CollisionManager

This is an abstract class with the intent that a game developer creates their own, game specific collision manager that derives from this one. This class is where the collision type enum is declared as well as a collision struct which holds a collision type, a colliding entity and another colliding entity. The manager itself holds a list of collisions called “collisionManifold”. The class has three important functions. First is “ProcessCollisions”. This method should be called in the main game loop every frame. Second, “CollisionBetweenCamera” which takes an entity and collision type as parameters. It loops through the collision manifold to check if the currently processed collision has already been detected, if it has, it will be ignored to avoid duplicate collision responses. The third method, “CollisionBetweenEntity” is just like the second, except this one takes a second entity as a parameter and checks the collision manifold accordingly.

EntityManager
This manager holds a list of Entity objects which can be updated by calling this class's "AddEntity" method. It also has a method that allows a developer to search the current list of entities for a specific entity, as well as a get function. The close method of this class calls the close method of all entities by looping through the list.

InputManager
This is another abstract class that is supposed to be derived from by the game developer using the engine. It holds an array boolean values, each boolean corresponding to a key on a keyboard. The key up method simply takes the currently pressed keys and sets the appropriate bool value to true, meaning this can be read to determine which keys are pressed. The key up method works in the opposite way, setting the bool values to false when a key is released. The class also contains abstract key up and key down methods, so that a game developer can implement their own logic in the event they need to know when a key has been specifically pressed down or pressed up, instead of just having access to the currently pressed keys.

ResourceManager
This class handles the loading and unloading of all game resources such as textures, shaders, models and audio. It is a static class, meaning it can be called without creating an instance of it. It works by storing each loaded resource in their respective dictionary (texture dictionary, audio dictionary, etc). This avoids duplicate resources being loaded.

SceneManager
This is an abstract class, meaning a game developer must create their own implementation of a scene manager. It holds numerous methods, such as keyboard input delegate methods, render and update delegates, as well as information about the window size and position. There are only three abstract methods in this class. "ChangeScene", "StartNewGame" and "StartMenu". This allows a developer to have more control over how a game switches scenes and begins its life in the main class.

SystemManager
The system manager holds a list of systems and the appropriate methods for adding to this list as well as finding specific entries in this list. Most importantly, it has a method called "ActionSystems" which is called every frame in the game loop, triggering the code in all systems in the list.

Class Descriptions – Engine Scenes
GUI
This is a static class, meaning it can be called from without creating an instance of it. It allows a developer to draw images and text on the screen by constructing a label through one of the many constructors. One constructor takes a string parameter, meaning this will create a text label on the

screen. Another takes a bitmap as a parameter, allowing a developer to draw an image label. The render method in this class must be called in the games render loop every frame.

IScene

This is an interface, meaning it is intended to be derived from by other classes. It is also where the scene types enum is declared.

Scene

This is an abstract class that derives from the previously mentioned scene interface. The intention here is that developers can create their own scenes that derive from this abstract scene. Each of those scenes must include a “Render”, “Update” and “Close” method.

Class Descriptions – Engine Systems

ISystem

This is another interface class that all system classes must derive from. It declares that all derived classes must include an “OnAction” method.

SystemAudio

This class includes a component types mask, consisting of two components, the position component and the audio component. This is a variable in all of the following systems class’s and each have their own unique form of this. The “OnAction” method of this system checks the mask of the current entity and compares it against the mask of the system. If the entity mask contains the components in the system mask, then the code block is executed, running the system. In this case, it checks if the entity has an audio component and a position component. If it does, it simply updates the audio position to the current position of the entity.

SystemCollisionCameraLine

The mask of this system requires that the entity must have a position component and a line collision component in order for the system to run. If that’s the case, then the line collision method will execute. This performs a collision check and, if true, calls the “CollisionBetweenCamera” method in the collision manager.

SystemCollisionCameraSphere

The mask of this system requires that the entity must have a position component and a collision sphere component. If so, the “Collision” method is called which checks for a collision between the camera and the sphere collider belonging to the current entity. Like before, if there is a collision, the appropriate method in the collision manager will be called.

SystemCollisionEntitySphere

This system has one mask that requires an entity has a position component and a collision sphere component. However, the "OnAction" method loops through all entities to find any that have the correct components. The result is two entities being passed to the collision method. This is because the system needs to check for a collision between two entities instead of one entity and the camera.

SystemCollisionLineSphere

This system has two masks. "LINE_MASK" and "SPHERE_MASK". Since the system needs to check for a collision between two entities, one of which having a line collider and the other having a sphere collider, two sets of components need to be checked for. If two entities are found with these components, then a collision check is performed against them. Again, in the event of a collision, the appropriate method in the collision manager is called.

SystemPathfinding

The pathfinding mask requires a position component and a pathfinding component. As previously mentioned, this isn't a "true" pathfinding system and instead acts more like an animation. The system will simply move the entity to the next point in the list, when it reaches the end of the list, it reverses the list and repeats the motion.

SystemPhysics

This system requires a position component and a velocity component. It simply updates the last position and current position of an entity appropriately.

SystemRender

This system requires a position component, geometry component and shader component. If so, it simply calls the "ApplyShader" method of the shader component and passes it the appropriate values.

Class Descriptions – Engine Objects

Camera

The camera class holds many methods for moving the camera in the scene. There are two constructors, one that takes no parameters, in which case a camera will be created at the default position. Another constructor takes a range of parameters that allows the developer to create a camera with their own custom properties

Entity

This class holds a string representing the name of the entity, a list of components and a component types mask. This mask is updated every time a component is added to the list.

Class Descriptions – Engine Misc

OBJLoader

The OBJLoader is a large collection of classes that handle the loading of data from .obj files.

Geometry

This class is a part of the OBJLoader and is the resulting object created from loading an .obj file. It holds all the loaded data such as textures and vertex information.

Class Descriptions – Game Managers

MazeEscapeCollisionManager

This class derives from the engine collision manager, showcasing how a developer can implement their own collision responses with the abstract “ProcessCollisions” method. This method is called every frame in the main game loop. It loops through all the collisions in the collision manifold and checks for specific conditions in the list of collisions, such as the names of the colliding entities, to determine the appropriate response.

MazeEscapeInputManager

This class derives from the engine input manager and showcases how a developer can implement their own input processing. The abstract “ProcessInputs” method from the parent class is implemented here to check for specific key presses in the array of pressed keys that is stored in the parent class. It also shows how a developer may want to implement their own specific “keyUp” event.

MazeEscapeSceneManager

This class derives from the engine scene manager and showcases how a developer can implement their own logic into how scenes are switched.

Class Descriptions – Game Scenes

GameOverScene / WinScene / MainMenuScene

These scenes all derive from the scene engine class, therefore must implement the “Render”, “Update” and “Close” methods. None of these scenes need any code in the update method, but they all share similarities in the code executed in the render method. The only differences being the text GUI displayed on screen.

GameScene

The game scene, again deriving from the scene engine class, is where the main game loop runs. This scene also holds the values needed to run game logic, is where entities and components are created, a long with the appropriate systems and managers needed.

Class Descriptions – Game Objects

Drone

The drone class holds all the information related to the drones in the game such as live remaining, the path they follow, the entity associated with it and the various bitmap images that represent the drone and its different states on the GUI.

Player

The player class is where the engines camera object is created as well as the appropriate player information such as lives remaining and weapon information.

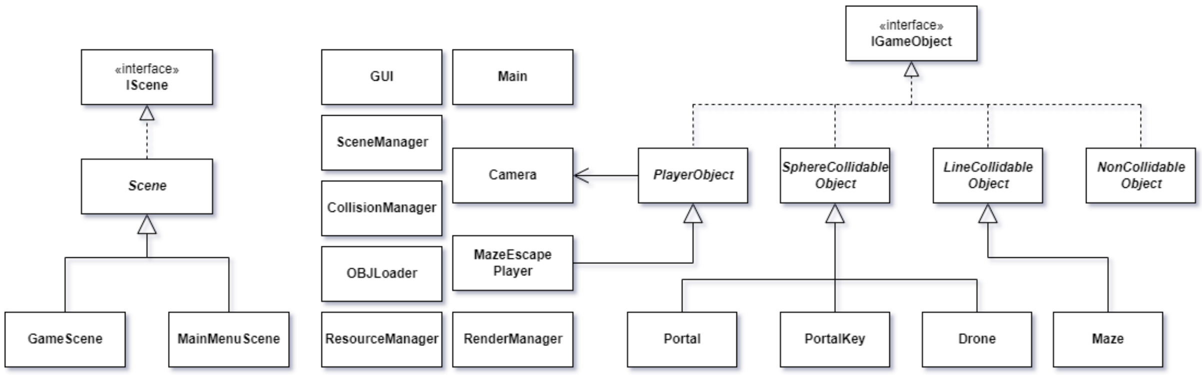
Main

This is the main entry point for the game. It simply creates a new game specific scene manager and begins the game loop from there.

Entity Matrix

Entity	**Position**	**Velocity**	**Geometry**	**Audio**	**ShaderDefault**	**ShaderNoLights**	**CollisionSphere**	**CollisionLine**	**Pathfinding**
Maze	x		x		x				
Floor	x		x		x				
DroneWhite	x		x	x	x		x		x
DroneRed	x		x	x	x		x		x
DronePurple	x		x	x	x		x		x
Health	x		x	x	x		x		
Emp	x		x	x	x		x		
WeaponUpgrade	x		x	x	x		x		
Skybox	x		x			x			
StandardBullet	x	x	x		x		x		
UpgradedBullet	x	x	x		x		x		
WallCollider(1-38)	x							x	

Inheritance-based design



The above image is my improved version of the inheritance-based engine design example. The main problem with the example given was the fact that a new class had to be created for every game object, even if a similar game object already exists.

For example, in the ECS design, you simply create a new instance of an entity and add the desired components to it. The systems associated with those components will handle the logic behind the components. If you wanted to create a "Portal" entity with a sphere collider and a "PortalKey" entity with a sphere collider. You can do so with just a few lines of code. However, with the inheritance design, you would have to create a class for the portal and another class for the portal key, of which you would create instances of. In addition to this, those two classes will most likely appear extremely similar, with only a few differences in the values stored in the class.

Whilst it may take longer to design and develop an ECS based game engine, that time and effort is something that will pay off in the long run. This is due to the modularity of its design. When it comes to actually developing a game with the engine, it is a lot easier to experiment with different ideas and implement gameplay elements because of the ability to create a new entity in just a few lines of code. With the inheritance design, there are a lot more hoops to jump through when creating a new idea and even scrapping a new idea. If an idea doesn't work out as intended, you could potentially be left with a number of now useless class files filling up the project which need to be cleaned out. And in the event that an idea warrants revisiting after previously being scrapped, those files would need to be created again. Furthermore, the modularity of ECS can boost creativity because of this saved time and effort when adding different components to an entity.

For example, take the question: "what would happen if we added sound to this portal?". In the inheritance design, the portal class would need to be edited. The code for loading a sound would need to be added to the constructor and the code for playing the sound and updating the sound location would need to be added to the portal update loop. On the other hand, with ECS, this same task could be completed with a single line of code:

portalEntity.AddComponent(new SoundComponent(portalSoundFile))

Furthermore, the abstractness of the ECS design is a large benefit over the inheritance-based design. By dividing the game logic into their own systems and components, it is much easier to identify any parts of the code that are causing problems. With inheritance, the source of a problem can be a lot harder to find due to game logic being in classes across the whole project or hidden by multiple layers of inheritance. ECS also makes sure that a fix in the physics system will apply to all entities with those physics-based components. Whereas with the inheritance design, the fixes will have to be copied across all classes that have physics-based properties.

Chapter 2: Evaluation

Choice of implementation

I chose to implement choice 3: "Compile your engine into a true library. The game will use this library rather than the source code of the engine."

To do this, I created a new DLL project in visual studio and copied over all the code I deemed necessary to the engine. Some changes had to be made, such as making classes public so that they can be accessed by the game project and correcting namespace errors. Initially, I wanted to store the shader files in the engine, but this doesn't work the way I wanted to with a dll file. Since the resource manager loads shader with a file path, those files would need to be included in the game files. So, the vertex and fragment shader files need to be placed in a "Shaders" file in the game project due to the fact that they will not be copied over with the dll file. Once this was done, I built the engine solution and located the output dll file ready to be added to the game. Then I went to the game project and removed all of the engine files, then proceeding to add the engine dll to the reference manager. I then had to go through all the game code to make sure it was referencing the engine library. The final result was a much simpler game development environment that wasn't cluttered or complicated by engine files.

AI Implementation

As previously mentioned, when detailing the classes of the engine, I did not manage to implement a true pathfinding system and instead settled for a pre-defined waypoint based enemy movement system that can more suitable be compared to an animation system than a pathfinding system.

The original plan was to implement the A* pathfinding algorithm by breaking the playable area down into a series of nodes. The pathfinding system would then run the A* algorithm on all entities with the appropriate components. The system would have two modes, "patrol" and "chase". In patrol mode, the AI would move between nodes in a pattern. If the player enters the detection radius of the AI, a check would be made to determine whether or not the player is in the line of sight of the AI. If it is, that specific AI agent would enter "chase" mode, in which their target would now be the player, choosing the node on the best path toward the player. If the player is closer than any surrounding nodes, the agent would move directly to the player until they touch the player, or the player becomes further away than any surrounding nodes. If the player were to remain over a certain distance away from the agent for a long enough amount of time, the agent would return to "patrol" mode.

Like detailed earlier, the pre-defined path system works by moving the drone to the next position in the list of positions. Once the end of the path has been reached, the list will be reversed and repeated again, causing the drone to move backwards along the path to where it started, where it will once again be repeated.

Game Production Evaluation

One, relatively small, issue I came across during development occurred when I tried rendering multiple images on the GUI. The game shows an image representing each drone and its status (if a drone has two lives, it will show an image of the drone with two hearts. If it has one life, it will show an image of the drone with one heart, etc). Originally, I had a series of checks in the render loop that would determine which of these images should be shown and create a new bitmap from the correct file. The same being done for showing the player lives, represented by hearts. This meant that anywhere between 3 and 6 bitmaps were being created and thrown out every frame which dramatically hurt performance. The fix was simple, create the bitmaps in the constructor and choose the correct bitmap in the render loop.

Now there was a big mess of Bitmap variables adding to the already growing mess of drone and player variables collecting at the top of the game scene class, at which point I decided to refactor these into their own class files to make the class more readable. This allowed me to make big improvements to the game logic in the rest of the game scene due to the ability to loop through a number of drone objects instead of referencing each pre-defined drone variable that was previously declared in the game scene class.