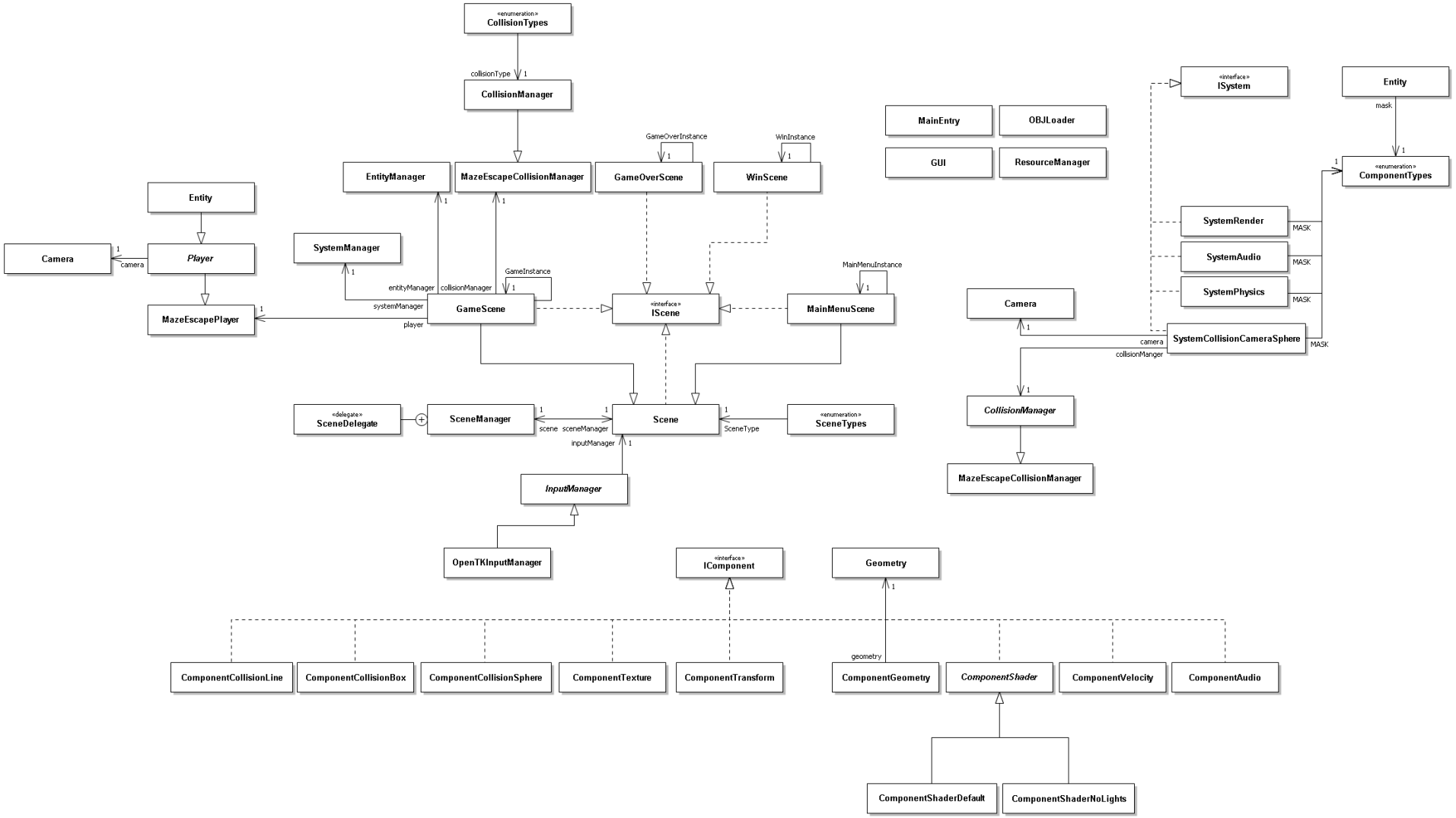
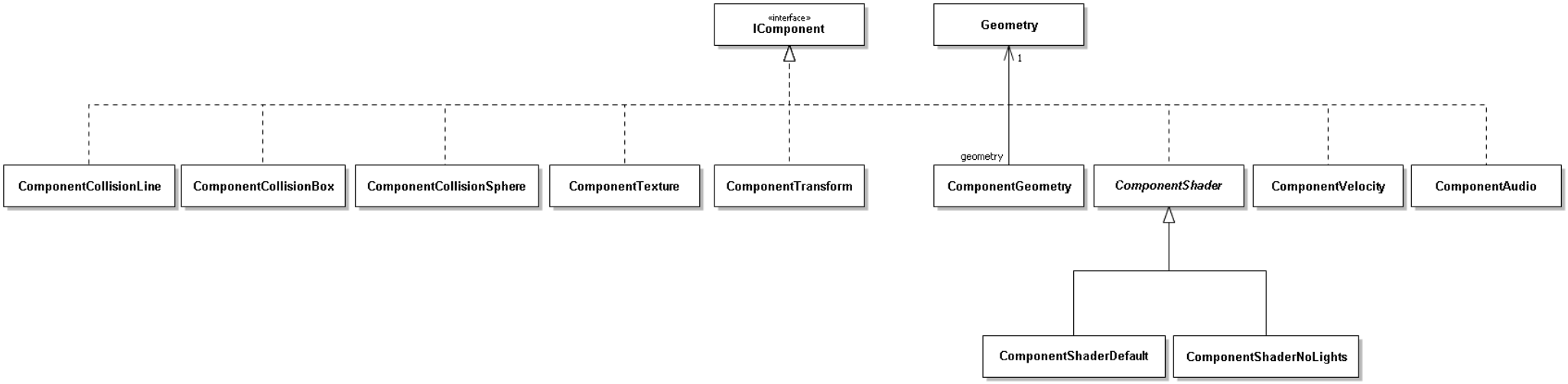
Game engine design and critique report

UML Diagrams

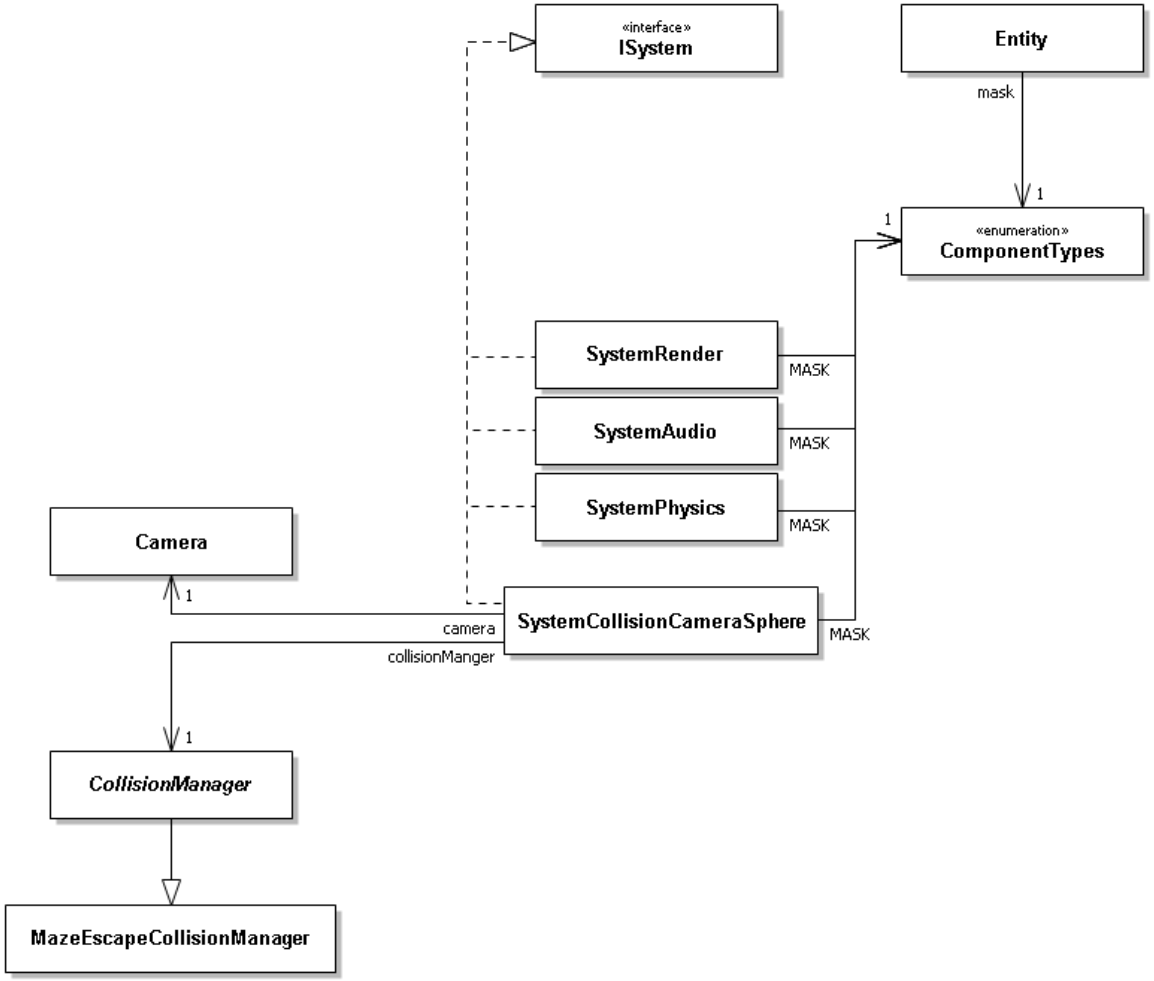
Overview



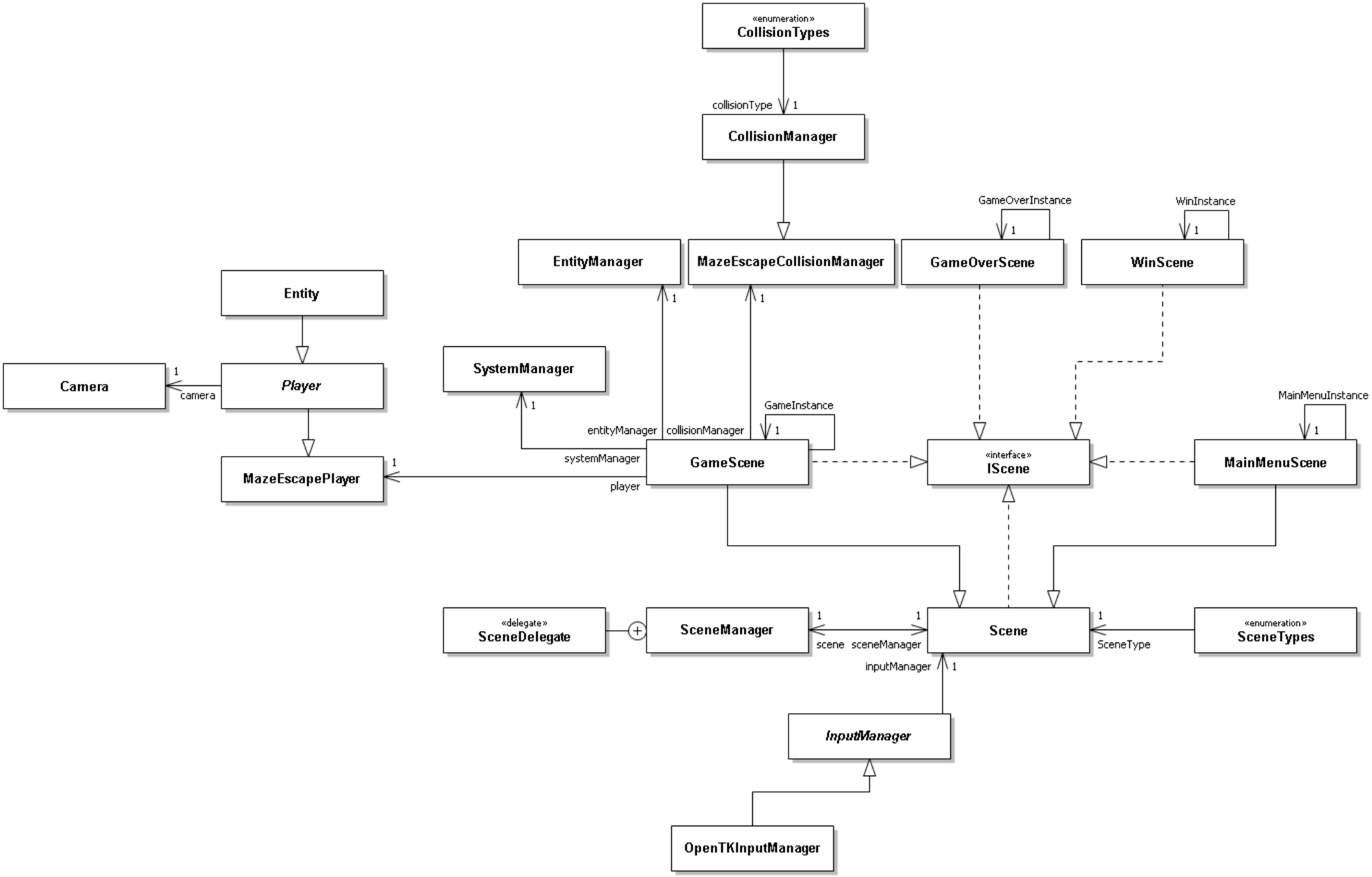
Component Overview



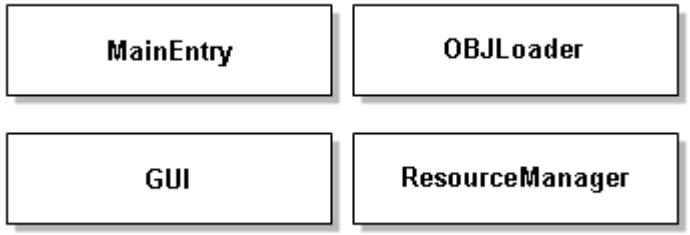
System Overview



Scene Overview



Singleton Overview



Descriptions and Justifications

Components

IComponent –

This is an interface class that all other components derive from, it ensures that all components must have a getter for the components type. The interface also contains an enumerator for component types.

ComponentAudio –

This component uses the audio file, that is passed through in the constructor and loaded by the resource manager, to set up an audio source. The component also allows the audio to be played and paused as well as have the emitter position changed using some methods.

ComponentCollisionBox –

This component creates a box for collision detection using points given by the constructor. Each point has its own getter & setter.

ComponentCollisionLine –

This component creates a line for collision detection using points given by the constructor. Each point has its own getter & setter.

ComponentCollisionSphere –

This component creates a sphere for collision detection using a radius given by the constructor. There is a getter & setter for the radius.

ComponentGeometry –

This component calls for the resource manager to load a geometry file that is passed in via the geometry component. The geometry that is returned by the resource manger is stored in the component and can be retrieved by a getter.

ComponentShader –

This component is an abstract class the forms the base for the other shader components. In the constructor the program ID is made using the passed in vertex and fragment shader files. A copy of the camera is also passed in for use in applying the shader. An abstract method for applying the shader is also in this component so all components the inherit from it must contain an override method for it.

ComponentShaderDefault & ComponentShaderNoLights –

These two components derive from the component shader class but both have identical code, the only difference being is what fragment shader is loaded. The constructors get the uniforms needed and store them in the component. The abstract apply shader method from the component shader is overridden and contains the appropriate code to load the shader onto the model and geometry that are passed into the method.

ComponentTexture –

This component loads the passed in texture file via the resource manager, the texture is then stored in the component. The texture can be retrieved using a getter.

ComponentTransform –

This component contains an entities position, rotation and scale. A copy of the position, rotation and scale is stored as the initials in case you want to reset an entity to its initial position, rotation and scale. The position, rotation and scale can all be retrieved and edited using their own getter & setter. The component also conations an identity getter which makes a matrix out of the position, rotation and scale, this is used when rendering the model.

ComponentVelocity –

This component contains a vector for the velocity of an entity. The velocity can be retrieved and edited using the getter & setter.

Managers

CollisionManager –

This is an abstract class that serves as a base for any collision manager. It contains all of the core elements that a collision manager will need such as a collision manifold. The class also includes an enumerator of collision types and a struct to define what a collision is.

EscapeGameCollisionManager –

This is the game specific collision manager that derives from the generic collision manager. This class defines what should happen when a collision happens between the camera and each specific entity.

EntityManager –

This manager is responsible for storing all of the entities as well as accessing these entities.

InputManager –

This is an abstract class that other input managers will derive from. It contains an abstract method for processing inputs so that all input manager will have to have an override for this method.

OpenTKInpuManager –

This is the specific input manager for the game, its override of the process inputs method defines what should happen when different keys or mouse buttons are pressed.

ResourceManager –

This manager is in charge of loading any resources and storing them in a dictionary. The manager also contains code to make sure things aren't unnecessarily load more than once, as well as a method to remove all the load resources.

SceneManager –

This class derives from the game window class that is apart of the open TK library. The manager controls the current scene as well as allows for the scene to change.

SystemManager –

This manager stores a list of all the systems in use and has a method to call all of the stored systems to action.

<u>Systems</u>

ISystem –

This class is an interface that serves as a base for all system classes. It outlines an on action method that all systems must implement as well as a name getter.

SystemAudio –

This class defined the on action method to make it set the audio components position to be the same a the entities position which is retrieved from the component transform.

SystemCollisionCameraSphere –

This system performs sphere collision calculations to determine if an entity has collided with the camera. If a collision has happened it is passed onto the collision manger.

SystemPhysics –

This system applies the motion for an entity based on the entities velocity component and the delta time.

SystemRender –

This system handles the drawing of each entity as well as applying the shaders for the entity.

Objects

Camera –

This class defines all of the information need to create the view for a 3D environment as well as methods to move the camera and retrieve and edit certain variables of the camera.

Entity –

This class is used to define entities in a scene. the class contains a list of components that the entity will have, this can be added to using a method once the entity is created. A mask of the components is stored as well as a name for the entity.

Player –

This is an abstract class that derives from the entity class and sets the name of the an entity of this type to be “Player”. This class also contains a copy of the camera since a player is generally directly affected or affects the camera.

MazeEscapePlayer –

This class derives from the player class and contains things that the player for this game needs. These things are a lives variable and a key card inventory variable.

Scenes

GUI –

This class sets up various methods for creating graphical interfaces for the user to interact with or to provide the user with information. The class also contains a method to render the graphical interfaces.

IScene –

This is an interface that serves as a base for the scene class. It contains an enumerator for scene types and some method outlines that will be required by all scenes.

Scene –

This is an abstract class that is the base or all scenes. It contains abstract methods for the methods that are required by the interface. The class also contains a public scene manager, a static input manager, a variable to store the scenes type and a static float that will serve as delta time.

DefaultScene –

This class derives from the scene class, the class overrides and sets up the appropriate method to create a simple blue scene with white text that displays "DEFAULT".

GameScene –

This class derives from the scene class and contains all the games logic as well as code to define all the required entities and all the appropriate managers and systems.

MainMenuScene –

This class is almost identical to the default scene the only differences being that the main menu scene initialises the input manager it inherits to an open TK input manager and that instead of the scene displaying "DEFAULT" it displays "MAIN MENU".

GameOverScene & WinScene –

These classes are identical to the main menu scene except the display the appropriate text depending on the scene.

Geometry

This contains two classes, the Group class and the Geometry class. The group class contains data of the geometry such as the VAOs and VBOs. The geometry class contains a list of group objects and the file path for the geometry that will be loaded. The geometry class also contains methods to load the object, load the OBJ object, render and remove geometry.

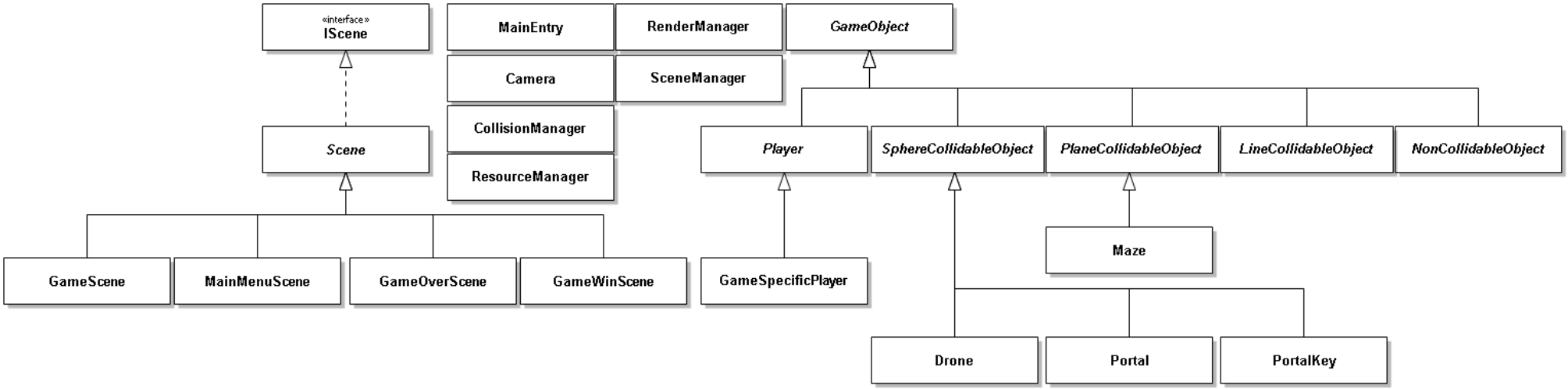
OBJ Loader

This contains all the necessary classes and code to be able to load data from a .OBJ file.

Entity Matrix

Entity	Audio	Collision	Geometry	Shader	Texture	Transform	Velocity	AI Control	Input Control
Player		X				X			X
Skybox			X	X	X	X			
Maze			X	X	X	X			
Key cards	X	X	X	X		X			
Portal OFF	X		X	X		X			
Portal ON	X	X	X	X		X			
Spike Pillar		X	X	X		X	X		
Spike Ball		X	X	X		X	X		
Drone	X	X	X	X		X	X	X	

Inheritance-based Design



Comparing my improved inheritance design to my ECS design it looks as if the ECS would more complex to use as the diagram is a lot bigger and has more classes and parts to it however that is because the ECS design is a lot more abstract allowing for the code to used for multiple games, whereas the inheritance design, while still has aspects of abstractness, is a lot more rigid and would require more editing to the base classes from game to game. While the inheritance design probably could be more abstracted than I have achieved it quickly becomes complex and hard to follow, so it would seem that with an inheritance based design you either have to sacrifice abstractness or simplicity. ECS design is definitely more modular due to its components whereas with an inheritance based design modularity seems as if it would be extremely hard to implement without the complexity getting out of hand quickly.