Team: David Baird
Aaron Holt
Kyle Olsen

Title: Asteroids 3D

Project Summary:

A game inspired by the 2D game asteroids brought into 3 dimensions.

Implemented and Unimplemented Features

We focused on implementing the actual gameplay over adding in optional features such as statistics, saved games, profiles and options. The sections of related requirements are listed in Table 1 along with a brief description of the implementation status for the features in that section.

Feature Implementation		
ID	**Requirement**	**Implementation Status**
U-1	Movement	All movement requirements were implemented.
U-2	Weapons	There is no secondary weapon. All other requirements were implemented.
U-3	Settings	The settings requirements were not implemented.
U-4	In-game Actions	You can exit the game, however you cannot pause or save.
U-5	Profiles	The profiles requirements were not implemented.
U-6	Statistics	The statistics requirements were not implemented.
F-1	Player Profiles	The player profiles requirements were not implemented.
F-2	Player Data	The player data requirements were not implemented.
F-3	Object Interactions	All of the object interactions except for the proximity warning were implemented.
F-4	Sound	The sound requirements were not implemented.
F-5	Movement	The movement requirements were not implemented.
F-6	Gameplay	The gameplay requirements were not implemented.
F-7	Viewing Angle	The viewing angle requirements were implemented.

Table 1: The categories of requirements and a brief descriptions of their implementation status.

As evidenced by Table 1, we focused upon features which directly related towards gameplay, which also encompassed most of the design work for the project. This reflects a change in priorities from the Part 2 document to heavily emphasize gameplay over any other feature. Overall, we implemented all critical features and constructed the setup for the menuing system. Thus the result is a playable and enjoyable game rather than a game which is feature heavy but not enjoyable. Despite not implementing some of the features, the design and implementation are extensible for those features.

Class Diagrams

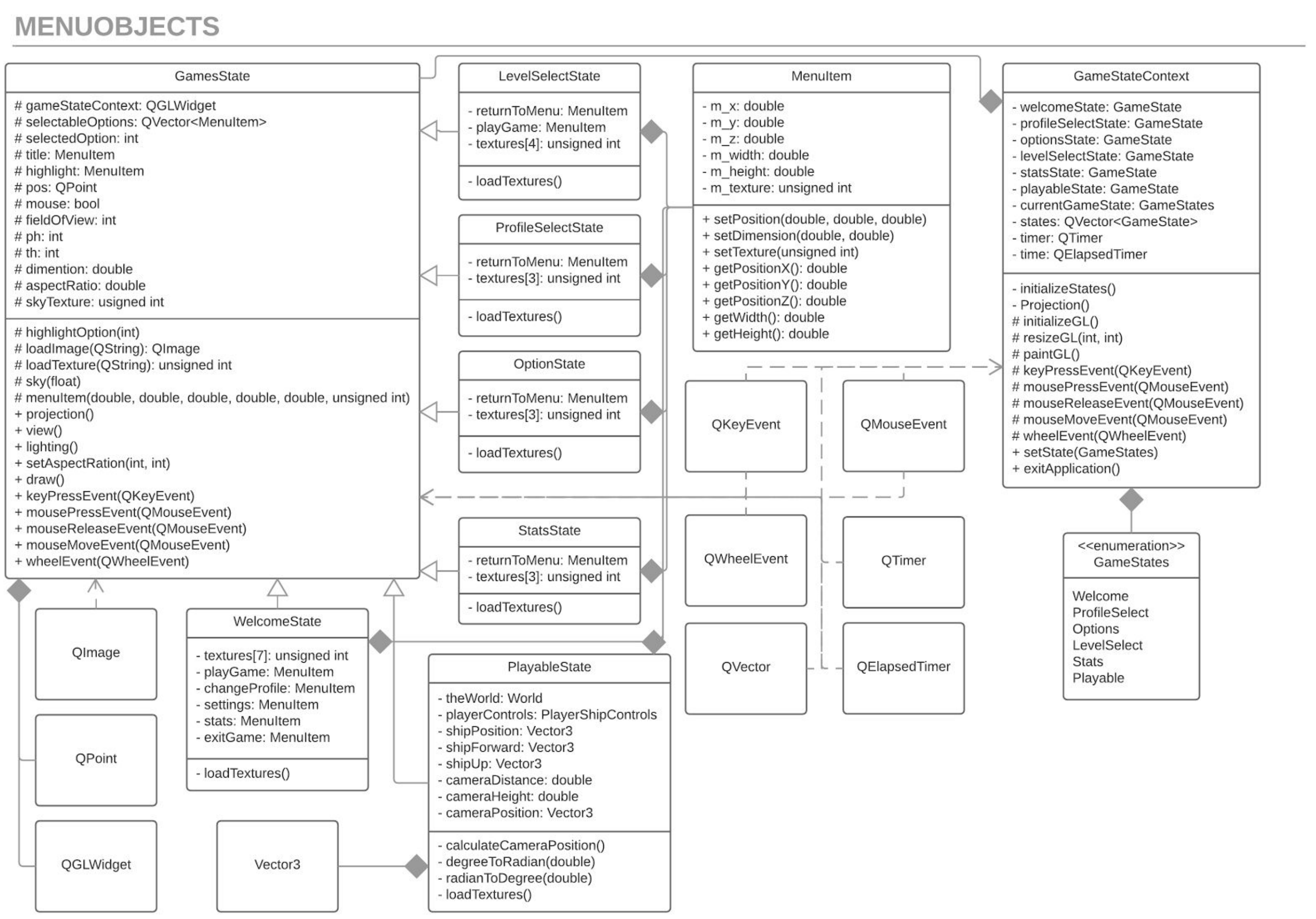

Figure 1: New Menu Class Diagram

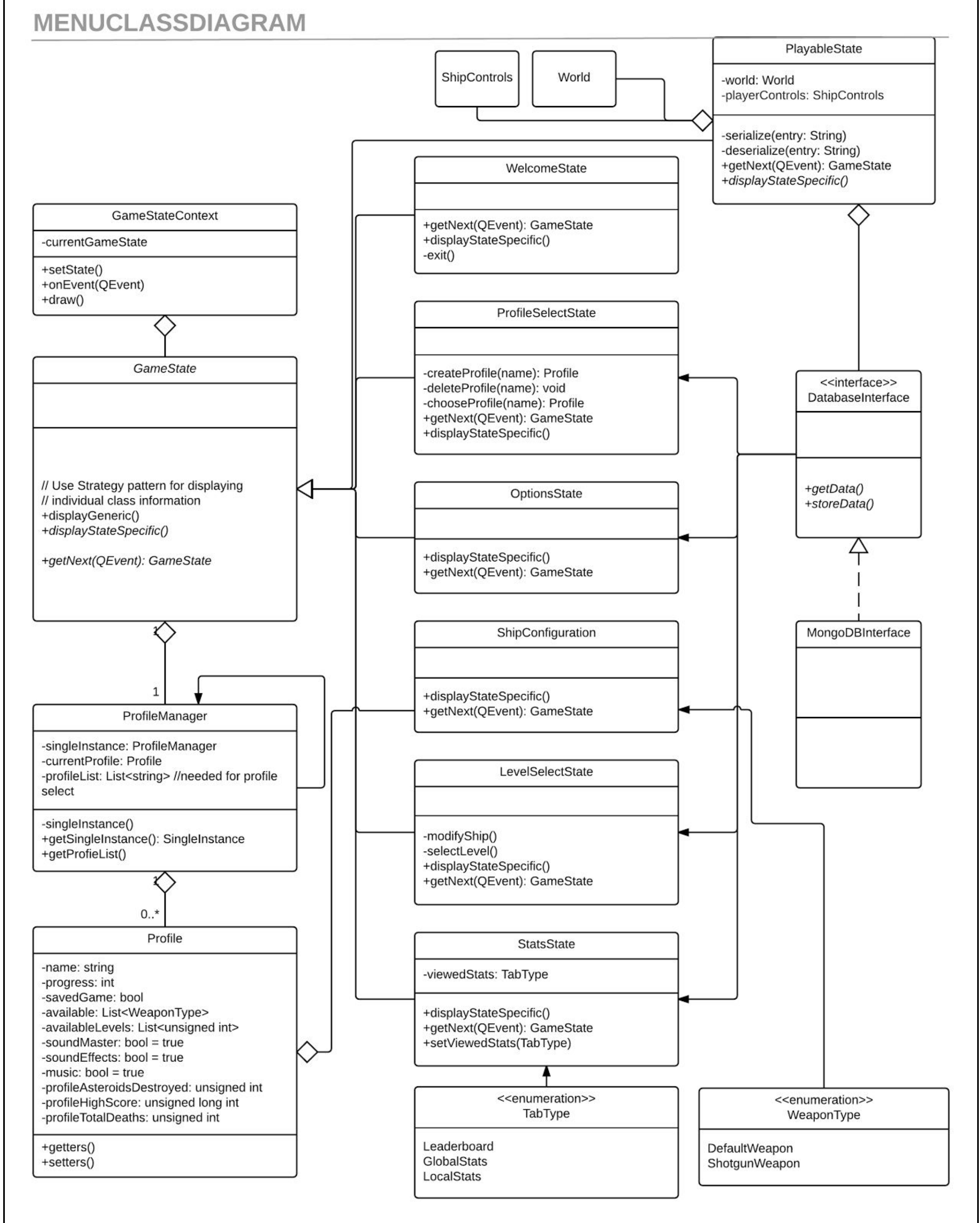

Figure 2: Original Menu Class Diagram

The biggest change to the menu diagram is the addition of a significant amount of QT related methods and variables. These updates were not in the original diagram because we lacked sufficient understanding of how QT would work with our application. The most notable additions are image and texture loading and methods for event handling. The profile elements present in the original diagram are not present in the present diagram because they remain unimplemented and unused at present.

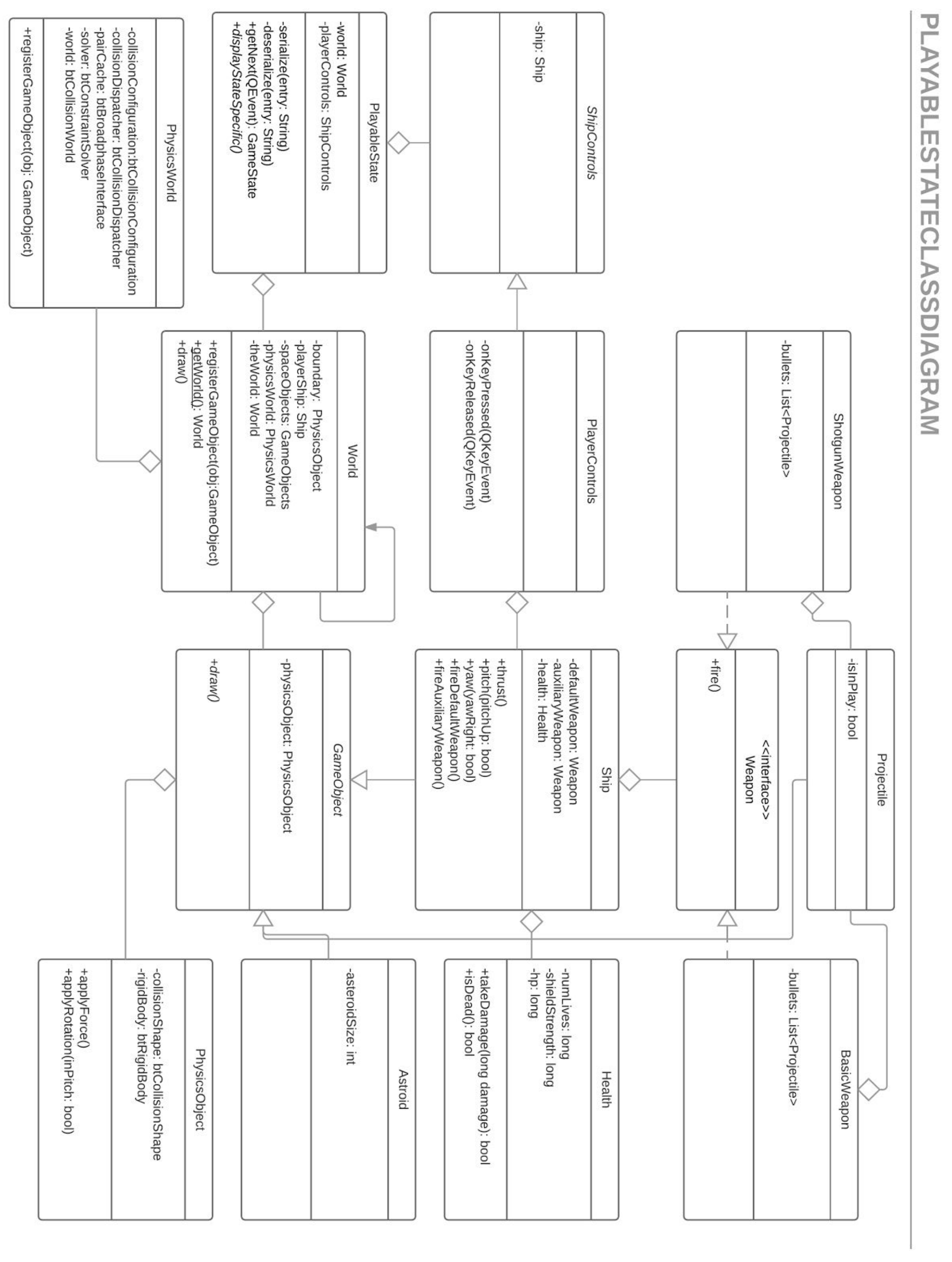

Figure 3: Original Playable State Class Diagram

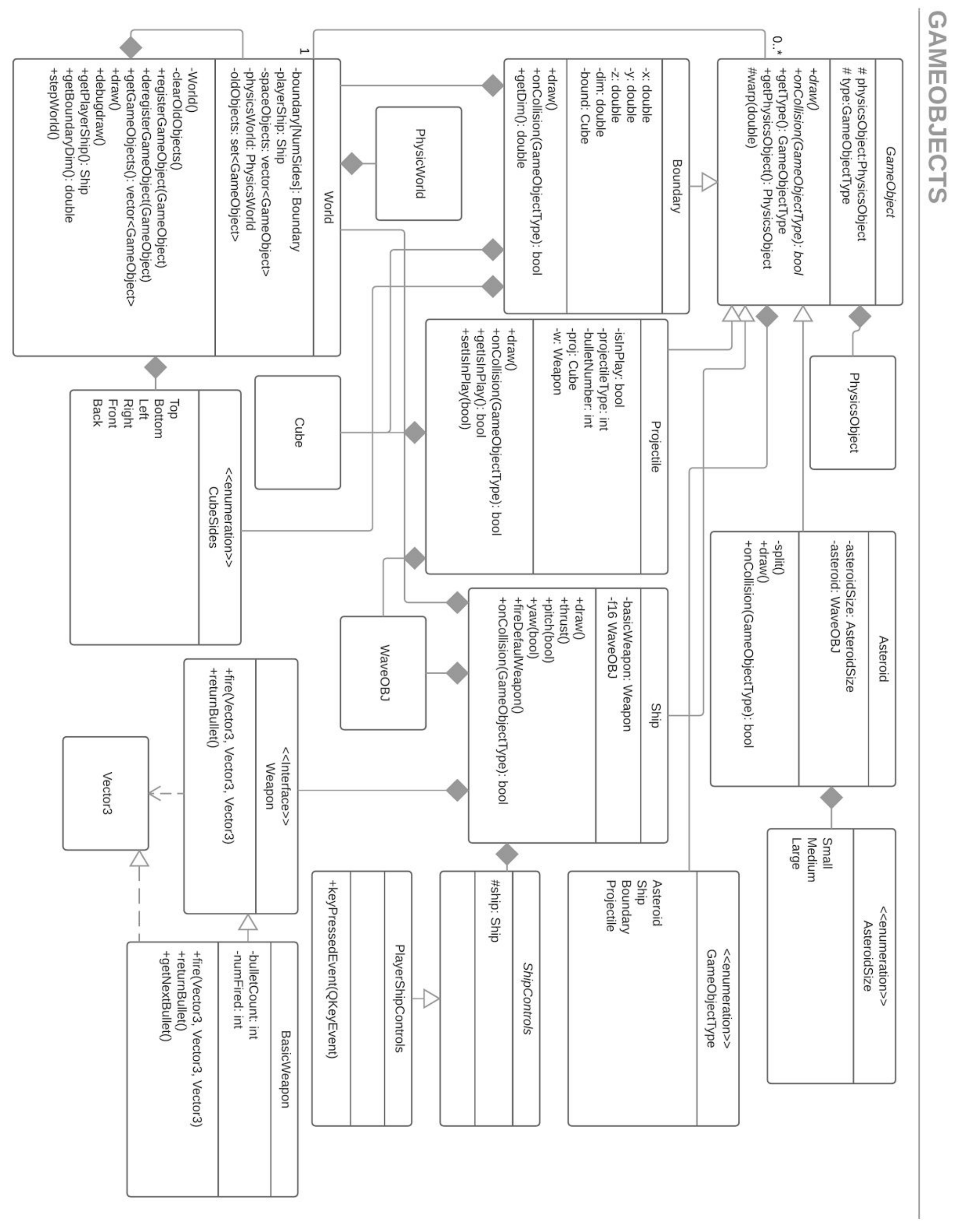

Figure 4: Game Objects Class Diagram

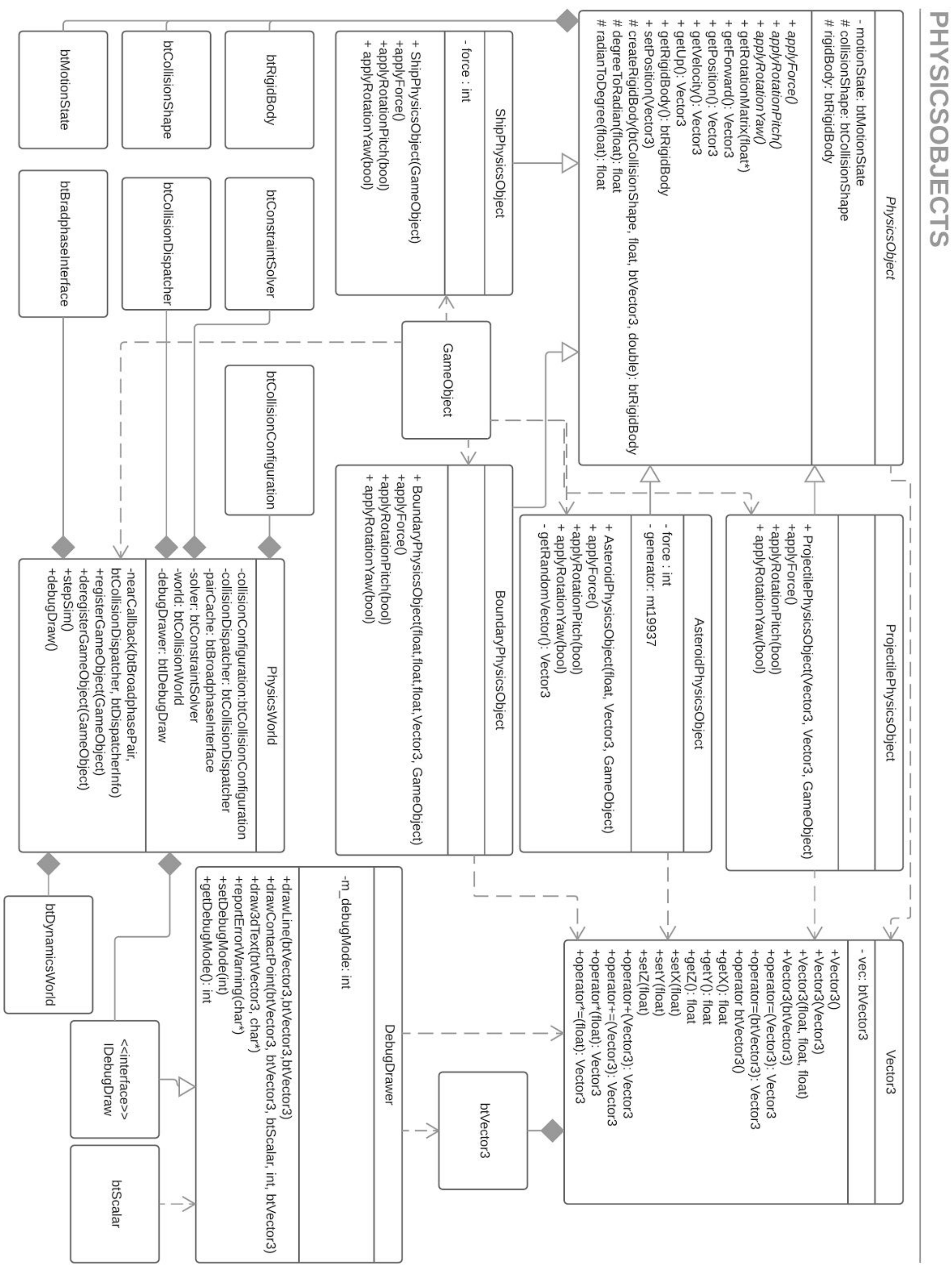
PHYSICSOBJECTS
PhysicsObject
- motionState: btMotionState
# collisionShape: btCollisionShape
# rigidBody: btRigidBody
+ applyForce()
+ applyRotationPitch()
+ applyRotationYaw()
+ getRotationMatrix(float*)
+ getForward(): Vector3
+ getPosition(): Vector3
+ getVelocity(): Vector3
+ getUp(): Vector3
+ getRigidBody(): btRigidBody
+ setPosition(Vector3)
# createRigidBody(btCollisionShape, float, btVector3, double): btRigidBody
# degreeToRadian(float): float
# radianToDegree(float): float
ShipPhysicsObject
- force : int
+ ShipPhysicsObject(GameObject)
+applyForce()
+applyRotationPitch(bool)
+ applyRotationYaw(bool)
ProjectilePhysicsObject
+ ProjectilePhysicsObject(Vector3, Vector3, GameObject)
+applyForce()
+applyRotationPitch(bool)
+ applyRotationYaw(bool)
AsteroidPhysicsObject
- force : int
- generator: mt19937
+ AsteroidPhysicsObject(float, Vector3, GameObject)
+ applyForce()
+applyRotationPitch(bool)
+ applyRotationYaw(bool)
- getRandomVector(): Vector3
BoundaryPhysicsObject
+ BoundaryPhysicsObject(float,float,float,Vector3, GameObject)
+applyForce()
+applyRotationPitch(bool)
+ applyRotationYaw(bool)
GameObject
btRigidBody
btCollisionShape
btMotionState
btConstraintSolver
btCollisionDispatcher
btBradphaseInterface
btCollisionConfiguration
PhysicsWorld
-collisionConfiguration:btCollisionConfiguration
-collisionDispatcher: btCollisionDispatcher
-pairCache: btBroadphaseInterface
-solver: btConstraintSolver
-world: btCollisionWorld
-debugDrawer: btIDebugDraw
-nearCallback(btBroadphasePair, btCollisionDispatcher, btDispatcherInfo)
+registerGameObject(GameObject)
+deregisterGameObject(GameObject)
+stepSim()
+debugDraw()
btDynamicsWorld
DebugDrawer
-m_debugMode: int
+drawLine(btVector3,btVector3,btVector3)
+drawContactPoint(btVector3, btVector3, btScalar, int, btVector3)
+draw3dText(btVector3, char*)
+reportErrorWarning(char*)
+setDebugMode(int)
+getDebugMode(): int
<<interface>>
IDebugDraw
btScalar
btVector3
Vector3
- vec: btVector3
+Vector3()
+Vector3(Vector3)
+Vector3(float, float, float)
+Vector3(btVector3)
+operator=(Vector3): Vector3
+operator=(btVector3): Vector3
+operator btVector3()
+getX(): float
+getY(): float
+getZ(): float
+setX(float)
+setY(float)
+setZ(float)
+operator+(Vector3): Vector3
+operator+=(Vector3): Vector3
+operator*(float): Vector3
+operator*=(float): Vector3

Figure 5: Physics Object Class Diagram

The playable state class diagram was split into two different diagrams: the physics objects class diagram and the game objects class diagram. This change is a result of splitting the physics object into seperate classes for each type of game object. This choice was made because a single physics object violated the single responsibility principle by having to handle the reactions for each type of game object. The creation of these classes created the logical distinction between game objects, responsible for drawing and other non-physics aspects, and the physics objects, handling all the physics components. Other changes between the diagrams are due primarily to a lack of foresight into what information the game objects would need from the physics objects.

Design Pattern

The project primarily features the state and observer design patterns. The state pattern is used in the menuing system to create a simple, extensible flow between the different screens. This design also simplified the handling of user input by making a simple consistent interface through which user events were uniformly processed until the appropriate state (screen) handled the event as appropriate for the context.

The observer pattern is used twice in the project. The first allows projectiles to inform the weapon that fired the projectile of its destruction. This use allows weapons, specifically, and other weapons, generally, to keep a maximum number of projectiles in play or other restrictions the weapon may have based upon the projectiles destruction. The second use has any object with physical interaction in gameplay register with the physics engine, then upon the physics engine detecting a collision between two objects we notify each object in the collision that it is colliding. Here the pattern allows for separation between the collision detection and correct handling of object collisions.

There are other patterns which we did not make use of that could be included in the future to further improve the game. The most important of these is the Flyweight design pattern. In the game the flyweight pattern would be used to avoid creating multiple objects which represent the same graphic model. Specifically the flyweight pattern would enforce the creation of a single asteroid model which would reduce both the application's memory footprint and improve the efficiency of creating asteroid objects.

Lessons Learned

Our experience designing and creating this project taught us a few lessons. First was that while the initial design and documentation takes a significant amount of time, it allowed our team to better collaborate because it gave us a reasonable expectation of how the interfaces between classes should appear. Second, as an application develops some changes to the design are necessary and should not be disregarded. As we learned more about QT and Bullet, we found the original designs needed revision and giving ourselves the flexibility to make those changes improved the cleanliness of the code. Finally due to our utilization of object oriented concepts, such as encapsulation, we were able to modify our design on the fly without modifications to disparate parts of the code.