Basic math classes and utilities

float3.h

float4.h

mat4.h

Renderer

Opengl renderer, using geometry shaders to generate vertices. Geometry shaders have a bad rep for being slow and rightly so, but with newer hardware this may not be true anymore. Thus, this renderer specifically targets newer graphics cards. Loosely inspired by https://factorio.com/blog/post/fff-251

The renderer is capable of rendering point sprites, colored/border quads, lines and text. The rendering is batched, and the current implementation only produces 4 draw calls. This is achieved using a texture array which all textures are loaded into. There is room for future improvements, specifically to reduce wastage in the texture array. To improve the texture array, one could implement texture atlasing.

Renderer.h

Batch.h

Camera.h

Color.h - just typedef of float4

FontLoader.h

LineVertex.h

QuadVertex.h

Shader.h

Sprite.h

Texture2D.h

TextVertex.h

VAO.h

VBO.h

Utilities

BasicRandom.h

BasicTimer.h

Profiler.h

InputManager.h

Added support for game controllers

Fixed bug with KeyDown/KeyUp

Made inputs available in global scope (perhaps not the best idea)

Gameobject model

Contigious arrays for each entity type

Reusing memory instead of new/delete

Fixed memory footprint, not the best solution but works well for smaller games.

Entity.h

EntityManager.h

The entity manager manages the lifetime of all entities in the game. For each entity type, the entity manager keeps an array. The user has to define the amount of a certain entity type should be used in the game, by calling InitializeEntityArray with a template parameter and whatever parameters are required for the object's constructor.

InitializeEntityArray, calls the constructor on all the objects which it creates. It creates them in a contiguous array. CreateEntity calls the Start function of the object which is going to be reused.

RemoveEntity calls the disable function.

EntityType.h

EntityType uses a neat trick with static variables and templates. Each time a new version of the function is defined, the counter is incremented. Essentially giving a unique id for each entity.

StateMachine

StateMachine implementation used for managing game state. Should be effective for general purpose StateMachine uses too.

Statemachine.h

IState.h

Game systems

The collision system uses a SpatialHashing implementation improve performance. The implementation is not very optimized in terms of the input data, a lot of checks are unnecessary due to checking if the

two objects should collide with each other. One optimization may be to change the data structure used to separate objects depending on what they collide with.

For calculating the hash, I used simd operations, mostly for fun. It basically just hashes all the 4 corners at the same time.

The actual collision testing uses intel's threading building blocks library (TBB), which implements a parallel for loop, to me this is one of the easiest ways to program in parallel. Because the spatial objects are in contiguous arrays, parallelizing the box test scales well. The Collision detection system can deal with a pretty good number of entities, tested 6000 entities. The bigger issue with performance is the game object model and “registering” collision objects.

CollisionSystem.h

Entities

Bullet.h

Basically, the LightBullet and DarkBullet are pointless classes since they have the exact same behaviour, the only difference is their entity type. Everything could have been in BaseBullet and there would be less code. At the time when I started working on them however, it was not clear if they would have the same behaviour.

Destroyer.h

The “boss” enemy of the game. A pretty generic enemy which just spews bullets around itself.

Player.h

PlayerWeapon.h

WeaponPickup.h

Meant to switch between different weapons, never got around to implementing it properly.