**Graphics Layer**

This document describes how the game draws its contents on the screen. All its subsystems are described on this document.

**Additional Primitives**

Those are the primitives implemented in our library:

* A vector in 2D with facilities to compute its length, dot, addition, rotations, angles between two 2D vectors and so on.
* A transform indetifiying a rotation, scale, translation and shearing. A way to concatenate transforms (?). A way to reverse transforms (?).
* A 3x3 matrix that can be computed from a transform. Ways to compute the determinant, inverse and maybe a transform (?).
* Rectangle: An AABB rectangle. But maybe we should have methods to deal with non-AABB rectangle (?). Method to convert to polygon. Apply transforms. (?)
* Ray: Implement a ray like in box2d. Maybe implement collision detection. Apply transforms.
* Circle: Collision detection (?). Converts to polygon. Apply transforms.
* Polygon: A set of points forming a closed or opened polygon. We’re going to need:
  + Apply transforms.
  + Automatically eliminates duplicated points.
  + Test the convexity of the polygon. Test if the polygon is simple. Test if the polygon is open or closed.
  + Convert to mesh (with subdivision level) applying a texture map.
  + Collision detection between other geometry types. (?)
  + Split the polygon by a line. The line is identified by two points in space. Actually remove part of the polygon. Can only be done in simple polygons.
  + Add Space: Join two polygons by mutating the first.
  + Get Bounding Box.
  + Add and Remove Vertices from the polygon.
  + Compute the convex hull.
  + Test if completely inside.
  + Outline. Only for simple polygon. Return the outline of the polygon passing a width and the type of corner (miter or bevel corner).
* A class to test collisions (?).

**Resource Management**

* Study the possibility of not using shared\_ptr.

**Printing System**

We’re going to implement a system where you can append lots of data to a string encoded as UTF-8. The following primitive types are supported:

* All primitive types.
* Support for enumerations and enumeration flags.
* Support for strings itself.
* Support for all c++ containers.
* Support for custom types.
* Support for shared\_ptr, weak\_ptr, unique\_ptr and auto\_ptr.

We’re also will support:

* Log: Message, Warning and Log.

**Triangle Batch**

This class batches meshes of triangle to the OPEN GL Renderer. The vertex format will have the following attributes:

* Position.
* Color.
* Texture Coordinates.

So we can have a unique class for vertex attributes. There is two types of triangle Batches:

* Dynamic: Those will be updated every frame. The batcher does not hold reference to the geometry it renders.
* Static: Those won’t be updated every frame. The batcher copies the geometry and does not hold any reference in cpu memory.

The batcher supports only one type of geometry:

* A Mesh: a collection of triangles (vertex and indices).
* Each mesh has the notion of position inside the triangle batch.

The batcher uses indexed triangle lists. The batcher also accepts the following parameters:

* A set of textures to be used and bind to the active textures.
* A program used to render the meshes. Let’s use a simple approach. We’re not going to do a generic program class. A program in this cases aggregates the uniforms values it uses.
* A blend mode.

**GLStateManager**

* Manages the state of Open GL. Basically this class caches Open GL State so you don’t need to set again states you’ve already setted. We assume that we will always work with the same OpenGL context. So we assume that the app never changes the OpenGL context.

**Mesh**

A mesh is a set of vertexes to be rendered. It’s not associated to a texture. The following special type of mesh exists:

* Quad Mesh.

We’re gonna have:

* Ways to paint to a mesh.

**QuadMeshCache**

* This quad mesh cache caches quad meshes. Basically you’ll always create meshes from this cache.
* This quad mesh cache will be used to particle rendering.

**TextureMap**

Describes how to apply a texture to a mesh.

**MeshGroup**

* This is a group of textured meshes or solid color meshes.
* Each Mesh can be assigned:
  + A texture or solid color to be associated with.
  + A program to be used with the mesh and the constant textures the program needs to execute. For example, a lighting map.
  + A blend mode.
  + A position inside this mesh group.
  + A transform associated with this mesh (position, scale, rotation). We’re not going to support generalized transforms.
  + The type of mesh: dynamic, static, static transformable.
  + The meshes are identified by pointer semantics, not by value.
* The mesh group has:
  + A transform associated with the entire group.
* The mesh group manages one or more batchers and does its best to batch as many calls as possible. The Z order is respected if possible. If you want to make sure an element is rendered on top of another, use more than one mesh group.
* A mesh group takes optionally a render target.
* Enable clipping or not. Clipping is only done in dynamic geometry. Its advisable to enable clipping only when you are dealing with large batches of dynamic geometry since clip forces the mesh group to compute things that would be done in the vertex shader. Clipping is done with the static geometry as whole (by using its associated AABB) or with static transformable geometry.

**Texture**

A texture is a rectangular pixel area. We’re going to have the following methods:

* Methods to return the width, height of the texture and pixel format.
* Method to load textures from files.
* The texture is not modifiable.

**Texture Processing**

A texture may have its contents modified. We’re going to use the Core-Image API (maybe not) to produce the final. If we use CoreImage, check if CoreImage changes the OpenGL context and restore back (if that’s the case). Each stage contains a set of operations. For example, a state may blend some colors and etc… The following stages are supported.

* The base color or base texture.
* Some texture with one of the following blend modes (and opacity):
  + Normal, Multiply, Screen, Overlay and Color.
  + Transformation (Rotation, Scale, etc…)
  + Wrap Mode.
* A color with one of the blend modes above.

You specify a size in pixels (if you want to resize) or the texture processing will infer a size. This is readonly. But we’re gonna have methods to save the modified texture to a file.

**Texture Atlas**

* A texture atlas gets a set of texture and distributes in an atlas.
* It does that dynamically based on a vector of textures you pass in.
* We’re gonna have methods to retrieve how the texture were arranged, to save that data and save the atlas itself.

**Texture Atlas Proxy**

* Used in some scenarios as a texture. Can be used to texturize

**Texture Manager**

* Manage textures. Basically it has methods to pin a texture to a memory (so its not reclaimed as free memory).
* Have methods to cache texture. In other words the texture is only reclaimed when the app receives a warning.
* Have methods to release texture data when the app enters background and restore it when the app returns.

**RenderTarget**

* Creates a render target. A render target is a texture you can render too.