The idea of these couple entries will be to describe how I load my renderer's pipeline from a YAML file. In this particular entry I'll be describing the renderer architecture as it is right now.

From YAML to renderer in 50ms
Part 1

Before I start let me say that I'll have to refactor a chunk of this whole thing for implementing shadow maps, they're not implemented yet, which will probably be an ugly patch until I iterate on it a couple of times. So better do this right now, and maybe do an update once I have shadow maps working someday in the uncertain future.

Overview

My current (OpenGL 3.3) renderer has these main abstractions involved:

- Render passes, which represent entire rendering passes, API state needed and shader programs involved.
- State descriptors, used to store the API state that each pass needs to work.
- Render tasks, which contain the necessary data to issue draw calls.
- Block updaters, which update uniform blocks, aka constant buffers, and place the data in the buffers appropriately.
- Render batch, which is in charge of drawing a bunch of tasks passed to it, and managing the logistics of the block updaters.

I'll be describing each as they appear.

The OpenGL 3.3 renderer implements this simple interface for now:

--- begin code ---
```
public interface RenderDevice
{
   void render ();

   void reloadPipeline ();

   void addToQueue ( RenderTask task );

   void viewFrustum ( ViewFrustum viewFrustum );

   ViewFrustum viewFrustum ();
}
```
--- end code ---

Have in mind this isn't contemplating different cameras (thus why a single viewFrustum setter) nor resource management. This is a purely drawing focused class. Hopefully in the not so far future I'll have another implementation of this interface for a Vulkan backend :3

Also that 'reloadPipeline()' is a plain hook to make the renderer reload all the render passes. I bind it to a key so I can reload it on the fly. As the title says, it takes roughly 50ms as long as no shader was modified.

By TheChubu, 18-07-2015

drawAllTheThings()

Since this is a journey through the renderer, its starts here, in the 'render()' function implementation of the OpenGL 3.3. renderer:

```
--- begin code ---
public void render ()
{
   setupFullscreenPasses();
   updateViewRays();
   updateViewFrustum();

   final Bag<RenderPass> passes = pipeline;
   final int pSize = pipeline.size();

   final Bag<InstancedBatch> batches = batchesPerPass;

   for ( int i = 0; i < pSize; ++i )
   {
      final RenderPass pass = passes.getUnsafe( i );
      final Batcher<?> batch = batches.getUnsafe( i );

      if ( pass != null )
      {
         pass.begin( allocator, rasterState, texUnits );
         OpsGL.checkError();

         if ( batch != null && batch.renderQueue.size() > 0 )
         {
            pass.render( batch );
            OpsGL.checkError();
         }
      }
   }

   for ( int i = batches.capacity(); i-- > 0; )
   {
      final Batcher<?> batch = batches.getUnsafe( i );

      if ( batch != null )
      {
         batch.renderQueue.clear();
      }
   }
}
--- end code ---
```

First 3 calls are pretty hardcoded mojo as you can see.

- 'setupFullscreenPasses()' adds a specific RenderTask to the predefined fullscreen batch instance.
- 'updateViewRays()' updates a global UBO that contains data to easily compute view space position for deferred rendering.
- 'updateFrustum()' updates another global UBO containing view frustum reltated data (width, height, ratio, inverse ratio, etc), useful for some passes.

By TheChubu, 18-07-2015

The important thing is that this implies that all fullscreen passes feed from the same batch, this will be relevant during loading of the pipeline.

Also another important bit is that the relationship between a “pass” and a “batch” is just by their position, if a batch is placed at the same index as a render pass, that render pass will be called with that batch.

There are 'null' checks in both since the render passes and batches weren't necessarily tightly added to the list, they were just assigned an index. Right now they're sequentially placed, so thats a bit of legacy.

Two things happen here, we call 'begin' on each RenderPass instance, then call 'render' passing the batch of the same index.

Oh you'll see this “BufferAllocator” (called simply 'allocator' here) passed around, it just a Java implementation detail, need ByteBuffers for passing data from/to OpenGL, and I made this BufferAllocator just to keep 'em tidy.

Let's see how RenderPass's 'begin' function looks like...

Further down the rabbit hole

--- begin code ---
```
public void begin ( BufferAllocator allocator, RasterState context, TextureUnit[] texUnits )
{
   // Source framebuffer.
   if ( source != null )
   {
      setupReadFramebuffer( texUnits );
   }
   // Target framebuffer.
   if ( target != null )
   {
      setupDrawFramebuffer( allocator );
   }
   // Blit buffer if necessary.
   if ( copying != null )
   {
      setupCopyFramebuffer();
   }
   // Setup descriptor state.
   setupDescriptors( context );
}
```
--- end code ---

This does 4 things, in order:

1. Sets up framebuffer's attachments for sampling if there is one attached.
2. Sets up framebuffer's attachments for drawing if there is one attached.
3. Sets up a framebuffer's attachment for copying (via blitting from one attachment to the draw one) if there is one attached.
4. Sets up the state descriptors added to that RenderPass instance.

TextureUnit is probably something I'd have to refactor someday. Its an array of that, texture units, so you can bind and set a sampler for them. They have a pretty fixed purpose, for example, TU from 8 and up are for sampling FBO attachments, TU 0 is for diffuse texture, TU 1 is for normals, and so on.

By TheChubu, 18-07-2015

Now the interesting one is RasterState, this one is passed to the state descriptors. RasterState instance contains a bunch of fields for different kind of OpenGL state for tracking purposes, a couple of its fields are like this for example:

```
--- begin code ---
// Triangle.
public int polygonMode;
public int frontFace;
public int cullFace;
public boolean faceCulling;

// Viewport.
public int viewportX;
public int viewportY;
public int viewportWidth;
public int viewportHeight;
--- end code ---
```

Each state descriptor gets the RasterState instance passed to it, so it can check the already set state, and modify it if necessary (which involves issuing the related OpenGL calls and updating the RasterState instance as needed).

All descriptors work in the same way: They have a Builder (yeah, the pattern) used for constructing a descriptor (also contains default values for all the descriptor's fields), they have this 'accept(rasterState)' method, and all their fields are public and final. Descriptors are immutable since they represent the state that the current RenderPass needs, unlike RasterState which represents the current ever changing state of the renderer pipeline.

I have descriptors such as StencilDesc, DepthDesc, ViewportDesc, etc. This is the 'accept(rasterState)' implementation of the ViewportDesc for example:

```
--- begin code ---
@Override
public void accept ( RasterState t )
{
   if ( t.viewportX != x || t.viewportY != y || t.viewportWidth != width || t.viewportHeight != height )
   {
      glViewport( t.viewportX = x, t.viewportY = y, t.viewportWidth = width, t.viewportHeight = height );
   }
}
--- end code ---
```

Depth, stencil, blend, and other descriptors have a more complex (longer!) logic for avoiding redundant state changes.

Thanks to these, RenderPass only needs to iterate over the descriptors it has, passes the RasterState instance to them, and they do their own state tracking job.

Now... draw!

Back to the top 'render()' function, next up is the 'render(batch)' implementation, this one simply calls the 'render(programs)' function of the batch, passing the shader programs the RenderPass has (usually only one).

The 'render(programs)' implementation depends on the batch type used. For now I only have the InstancedBatch subtype. This one does... complex things .

By TheChubu, 18-07-2015

--- begin code ---

```
public void render ( final Bag<ShaderProgram> programs )
{
   if ( renderQueue.isEmpty() )
   {
      return;
   }

   final UniformBlockUpdater<RenderTask>[] updaters = blockUpdaters.data();
   final int upSize = blockUpdaters.size();
   final RenderTask[] queue = renderQueue.data();
   final int quSize = renderQueue.size();
   final ShaderProgram[] progs = programs.data();
   final int prSize = programs.size();
   final int mbs = maxBatchSize;

   for ( int p = 0; p < prSize; ++p )
   {
      progs[p].bind();

      int taskOffset = 0;
      int instOffset = 0;

      do
      {
         int maxItem = quSize;
         int maxInstance = queue[quSize - 1].instanceCount;

         for ( int u = upSize; u-- > 0; )
         {
            final long encOffsets = updaters[u].batchUpdateInstanced
                                    ( queue, quSize, taskOffset, instOffset, mbs );

            maxItem = unpackHigh( encOffsets );
            maxInstance = unpackLow( encOffsets );
         }

         final int lastItem = maxItem - 1;

         // If the updater couldn't advance past the current item.
         int baseInstOffset = (lastItem == taskOffset) ?
                              maxInstance : queue[taskOffset].instanceCount;
         baseInstOffset -= instOffset;

         // Special case: First task draw.
         draw( queue[taskOffset], 0, baseInstOffset );

         for ( int i = (taskOffset + 1); i < lastItem; ++i )
         {
            final int ic = queue[i].instanceCount;
            draw( queue[i], baseInstOffset, ic );
            // Offset for next task.
            baseInstOffset += ic;
         }
```

By TheChubu, 18-07-2015

```
            // Special case: Last task draw.
            if ( (taskOffset + 1) <= lastItem )
            {
               draw( queue[lastItem], baseInstOffset, maxInstance );
            }

            final int lastItemCount = queue[lastItem].instanceCount;
            // Completed drawing/uploading instances of last task.
            final int completedLastTask = maxInstance < lastItemCount ? 1 : 0;
            // If yes advance to next item, keep it as it is otherwise.
            taskOffset = maxItem - completedLastTask;
            // Wrap over offset so next item starts from zero instances.
            instOffset = (maxInstance % lastItemCount);


         // While we dont run out of tasks.
         } while ( taskOffset < quSize );
      }
   }
--- end code ---
```

Yes! That same loop I quoted in a post about how horrible it was. Its awful. So I'll do some more understandable pseudocode instead:

```
--- begin code ---
for ( allPassedShaderPrograms)
{
   program.bind();

   do
   {
      for (allBlockUpdatersInThisBatch)
         update(tasks);

      for (allTasksThatCouldBeUpdated)
         draw(task);

   } while (thereAreMoreTasksToDraw);
}
--- end code ---
```

The important bit from here is that “block updater” thingy. An updater is basically an object to which you can pass a bunch of render tasks, and it will fetch the data it needs from them and put it nice and sequentially on an uniform buffer (aka constant buffer) for the GPU to access.

Each updater updates one particular kind of buffer. And since they have different size requirements, they can only update as many tasks as the biggest updater allows for.

This is often the Transform buffer, since it uploads two mat4s. So the maximum batch size of a RenderPass that uses the Transform updater would be 'uboCapacity/transformStructSize', thats often 64Kb and 64 bytes respectively, which gives you enough space for 1024 instances that use it.

That could be say, one render task with 1024 instances of the same mesh, two tasks with 512 each, or 1024 tasks of one instance each, you get the idea.

Once we're done with the uploading, we can issue an instanced draw call for each of the tasks that could be

updated. That means checking for cases like, uploaded all instances of task 1, but only got to half of task 2, and in the next iteration, pick up from where task 2 was left, then upload what it can of task 3, and so on. Thats why the uploading loop is so complex.

Draw logic looks like this:

```
--- begin code ---
private static final void draw (
   final RenderTask task,
   final int baseInstance,
   final int maxInstances )
{
   GL30.glBindVertexArray( task.meshId );
   final int[] texParams = task.texParams;

   if ( texParams != null )
   {
      for ( int i = 0; i < texParams.length; ++i )
      {
         final int params = texParams[i];

         int slot = OpsGL.unpackTexSlot( params );
         int target = OpsGL.unpackTexTarget( params );
         int texId = OpsGL.unpackTexId( params );
         GL13.glActiveTexture( GL13.GL_TEXTURE0 + slot );
         GL11.glBindTexture( target, texId );
      }
   }

   if ( task.indicesOffset > -1 )
   {
      DrawType.indexedInstanced( task, baseInstance, maxInstances );
      return;
   }

   DrawType.arraysInstanced( task, baseInstance, maxInstances );
}
--- end code ---
```

Basically, if there is any textures, unpack their parameters and bind them. If the offset into the index buffer is some valid index, then it means it needs an indexed draw call, otherwise do regular array call. All draws are instanced draw calls, even if the task has only one instance for drawing.

Now remember when I said one “block updater” updates one kind of buffer? Well it works like that... but it doesn't. They all use slices of a single shared big buffer as a sort of ring buffer.

Instead of having, say, ten 64Kb buffers rebound and updated, I only issue a new glBindBufferRange call each time, binding a slice of a 1Mb buffer, updating that, then moving onto the next updater and the next slice of the buffer, wrapping around as needed.

I only have three uniform buffers in the entire renderer actually, and I'm only using two at the moment, whereas in the shaders I have 6 different kinds of uniform buffers defined, they're all fed from the same buffer by simply binding different ranges of it to different constant buffer slots.

Now... I never described what is a RenderTask didn't I?

By TheChubu, 18-07-2015

RenderTask, how do they work!?

Its fields look like this:

--- begin code ---
public final byte type;
public final int meshId;
public final int instanceCount;
public final int vertexCount;
public final int baseVertex;
public final int indicesOffset;
public final int primitive;
/**
* Encoded texture params.
*
* 0xFF_00_00_00 tells you the slot.
 0x00_FF_00_00 tells you the
* binding target.
 0x00_00_FF_FF tells you the texture ID.
 Decode
* using {@link OpsGL} functions.
*
**/
public final int[] texParams;
// Arbitrary data.
final Object[] resources;
final byte[] resourceTypes;
--- end code ---

Field by field:

'type' is a predefined type to bucket up the render tasks. I got types like TYPE_MESH_STATIC or TYPE_MESH_ANIMATED. Those are used by the systems that create tasks, they grab entities from the game, create tasks from them, and submit them according to one of those types.

'meshId' is just the ID of the mesh being used, in this OpenGL specific case its a Vertex Array Object (VAO) ID.

Then you get the normal stuff like 'instanceCount' to draw of the same mesh, 'vertexCount' of the mesh, 'primitive' type to use (triangles, triangle strips, etc), 'baseVertex' that tells a vertex offset into the vertex buffer where the mesh starts, 'indicesOffset' which is an offset (in bytes!) into the index buffer where the indices of the mesh start.

Tasks, assemble!

The fun thing here is 'Object[] resources' which like an array of 'void*' in Javaland. Its an array of pointers to arbitrary objects.

Each task can be assembled with an arbitrary set of resources. Say, a point light task might have a model view transform, model-view-projection transform, and light a parameters object.

Each task also has an amount of resources of said types equal to the amount of instances it has. So if you have 100 instances for drawing in the task, and you know the task has transforms, it'll have 100 transforms.

This, in conjunction with 'byte[] resourceTypes', works as a tiny table. 'resourceTypes' is just an array of the type of resources in order of appearance.

So say that we have 50 instances in our point light task, we will have a resource type array like [RES_MODELVIEW, RES_MVP, RES_LIGHT_POINT]. This says to us that there are 50 model view matrices, 50 mvp

matrices and 50 point light parameter objects in the resource array, in that order.

Here is what the resource fetching function looks like:

```
--- begin code ---
public final <T> T resource ( final byte type, final int index )
{
   final Object[] rs = resources;
   final byte[] ts = resourceTypes;

   for ( int i = ts.length; i-- > 0; )
   {
      if ( ts[i] == type )
      {
         return (T) rs[(i * instanceCount) + index];
      }
   }
   // No resource of the given type was found.
   return null;
}
--- end code ---
```

Its not terribly efficient (linear search!) but the 'resourceType' array is really tiny, like 4 or 6 bytes tops, one byte per present type in the 'resource' array. I could probably encode it in a 'long' field or two if it ever becomes a problem, so that helps.

For example, this is how a static mesh task could be constructed:

```
--- begin code ---
MeshInstance mesh = [entity's mesh];
Material mat = [entity's material];

RenderTask task = RenderTask
                        .builder()
                        .type( TYPE_MESH_STATIC )
                        // This sets buffer offsets, mesh id, vertex count, etc.
                        .mesh( mesh )
                        .instanceCount( 1 )
                        // This one determines which textures all the instances of this task will use.
                        .material( mat )
                        .resource( mvpTransform, RES_MVP )
                        .resource( mvTransform, RES_MODELVIEW )
                        // This one is passed just for fetching the glossiness factor.
                        .resource( material, RES_MATERIAL )
                        .build();
--- end code ---
```

That task has only one instance, but if you wanted to, you could create a task for a hundred instances of that mesh and same material, by passing 100 resources of each kind the mesh needs to be drawn.

The intersting bit is that the “submitters” (systems that compose render tasks) don't need to know what API they're working with to compose rendering tasks, they just stuff the data in the tasks and queue them via the renderer interface. I hope is a good enough abstraction for implementing a different backend in the future.

By TheChubu, 18-07-2015

So... Um... Loading?

Well that was the rendering structure and process, next up is the loading of this whole thing, that will be in the next entry :D Cya!

By TheChubu, 18-07-2015