# Nublada

Thursday, November 2, 2023

8:36 AM

## GUI Rendering

Nublada renders their crosshair like so (note that these are legacy draw calls, so if they are not supported I am ok with that):

private void renderCrosshair() {

glUseProgram(0);

glColor3f(1.0f, 1.0f, 1.0f);

glLineWidth(2.0f);

glBegin(GL\_LINES);

glVertex2f(-0.1f, 0.0f); // Horizontal line (left)

glVertex2f(0.1f, 0.0f); // Horizontal line (right)

glVertex2f(0.0f, -0.1f); // Vertical line (bottom)

glVertex2f(0.0f, 0.1f); // Vertical line (top)

glEnd();

}

The cursor box is also generated likewise:

public void drawCube(float x, float y, float z, float size) {

// Set the color for the outline (white in this case)

glColor3f(1.0f, 1.0f, 1.0f);

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE); // Set the polygon mode to draw lines

glBegin(GL\_QUADS); // Begin drawing quads (faces of the cube)

// Front face

glVertex3f(x - size / 2, y - size / 2, z - size / 2);

glVertex3f(x + size / 2, y - size / 2, z - size / 2);

glVertex3f(x + size / 2, y + size / 2, z - size / 2);

glVertex3f(x - size / 2, y + size / 2, z - size / 2);

// Back face

glVertex3f(x - size / 2, y - size / 2, z + size / 2);

glVertex3f(x + size / 2, y - size / 2, z + size / 2);

glVertex3f(x + size / 2, y + size / 2, z + size / 2);

glVertex3f(x - size / 2, y + size / 2, z + size / 2);

// Connect faces to create the cube

glVertex3f(x - size / 2, y - size / 2, z - size / 2);

glVertex3f(x + size / 2, y - size / 2, z - size / 2);

glVertex3f(x + size / 2, y - size / 2, z + size / 2);

glVertex3f(x - size / 2, y - size / 2, z + size / 2);

glVertex3f(x - size / 2, y + size / 2, z - size / 2);

glVertex3f(x + size / 2, y + size / 2, z - size / 2);

glVertex3f(x + size / 2, y + size / 2, z + size / 2);

glVertex3f(x - size / 2, y + size / 2, z + size / 2);

glEnd(); // End drawing

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL); // Restore default polygon mode

}

# 

# Chunk generation

## The addChunk method

* Creates a new chunk
* Adds it to the HashMap
* Sends it to generationService, a ScheduledExecutorService that loads the chunk on a separate thread
* Stores the returned future object from the service execution onto a future list

## worldRenderer.render()

* **World.checkGeneratingChunks()**
  + iterates over the future list for all "loaded" chunks.
  + If a chunk is loaded
    - Chunk.initializeChunk()
      * chunk.generateMesh()
        + Submits a meshBundle to a meshService, an executor service. The mesh bundle computes the mesh and returns meshFuture. when done, the mesh will be sent to gpu within chunk.prepare on the render thread
      * Iterates over the existing neighboring chunks and marks them as not updated

* **if the player has moved,** 
  + mark needs sorting to true

* **Checks if any chunks around the player are non-existent and adds them using world.addChunk()**

* **If needs sorting**
  + Clear chunksToRender list
* **updateChunksToRenderList()**
  + Iterate over each chunk in the world
    - Chunk.prepare()
      * If not updated and loaded
        + Generate mesh again
      * If mesher future is not null and is done
        + Send mesh to GPU
        + Set mesher future to null
    - If the chunk is too far from the view distance, delete it
    - Otherwise, If needs sorting, add the chunk to the render list
* **If needs sorting**
  + Sort chunksToRender by distance from player
  + Set needs sorting to false

* **Render all solid meshes**
* **Render all transparent meshes**

//= new ScheduledThreadPoolExecutor(1, r -> { ... });: This line creates an instance of ScheduledThreadPoolExecutor.

//It's a type of ScheduledExecutorService that uses a pool of threads to execute tasks.

//- 1 specifies that the pool will have one thread. This means it will be capable of executing one task at a time.

//- r -> { ... } is a lambda expression that provides a ThreadFactory to the executor. It defines how threads are created. In this case,

//it creates a new thread, sets its name to "Generation Thread", and marks it as a daemon thread (meaning it won't prevent the JVM from exiting).

    private static final ScheduledExecutorService generationService = new ScheduledThreadPoolExecutor(1, r -> {

        Thread thread = new Thread(r, "Generation Thread");

        thread.setDaemon(true);

        return thread;

    });

    /\*\*

     \* The service.submit () method does not necessarily run the task on a new thread,

     \*  but rather on one of the threads in the pool managed by the ScheduledExecutorService.

     \*/

    Future<Chunk> future = generationService.submit(() -> {

        //Load the future chunk if it exists

        chunk.loadChunk(this, futureChunk);

        return chunk;

    });

**If a block is set on a chunk that does not exist, it goes to a future chunk.**

**If a new chunk is added, or the player moves, change the order at which the chunks are rendered**

Nublada greedy meshing and vertex data

Sunday, November 5, 2023

9:18 AM

* no voxel face class, just block ID
* No computing the side
* Instead of having one big mask, we make a new one for every new dimension
* When retrieving the voxel face, if it is transparent, set it to null

* IMPORTANT:

instead of

mask[n++] = ((voxelFace != null && voxelFace1 != null && voxelFace.equals(voxelFace1)))

? null : backFace ? voxelFace1 : voxelFace;

when adding to the mask, we do (The difference is that there is no block comparison):

mask[n++] = ((voxelFace == null || voxelFace1 == null))

? backFace ? voxelFace1 : voxelFace : null;

* when making the quad, there are 8 verticies instead of 4, making it a vertex and uv list.

\* every even vertex is a position, every odd vertex is a uv coordinate

\* the 3 coordintates of the verticies and uvs are compacted into one

\* each uv is a 3d coordinate U,V,layer

* the "vertex" list (that contains the vertex and uv data every other item) is passed to the GPU as a VBO of size 2, meaning that every 2 values correspond to 1 vector.

public int loadToVAO(int[] positions, int[] indices, int size) {

int vao = createVAO();

storeDataInAttributeList(0, size, positions);

bindIndicesBuffer(indices);

return vao;

}

private void storeDataInAttributeList(int attributeNumber, int size, float[] data) {

int vboId = GL30.glGenBuffers();

vboList.add(vboId);

GL30.glBindBuffer(GL30.GL\_ARRAY\_BUFFER, vboId);

GL30.glBufferData(GL30.GL\_ARRAY\_BUFFER, data, GL30.GL\_STATIC\_DRAW);

GL30.glVertexAttribPointer(attributeNumber, size, GL30.GL\_FLOAT, false, 0, 0);

GL30.glBindBuffer(GL30.GL\_ARRAY\_BUFFER, 0);

}

* The compactified coordinates are decompressed in the vertex shader:

in uvec2 vertex;

float y = (vertex.x >> 18u) & 0x1FFu;

float x = (vertex.x >> 9u) & 0x1FFu;

float z = (vertex.x) & 0x1FFu;

uint yUv = (vertex.y >> 18u) & 0x1FFu;

uint xUv = (vertex.y >> 9u) & 0x1FFu;

uint zUv = (vertex.y) & 0x1FFu;