Qt World Summit 2017

October 10-12 | Berlin, Germany

Earth Rendering With Qt3D

Paul Lemire

KDAB

Generating the Earth

1) What do we need?
    1) Generating the Earth Ellipsoid
    2) Adding Details with imagery tiles
    3) Terrain Elevation
2) In practice
    1) Drawing
    2) Building Data to be rendered
    3) Handling Precision issues

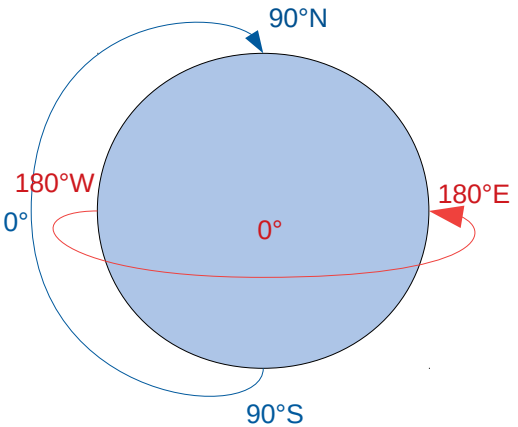
What do we need?

Qt World Summit 2017

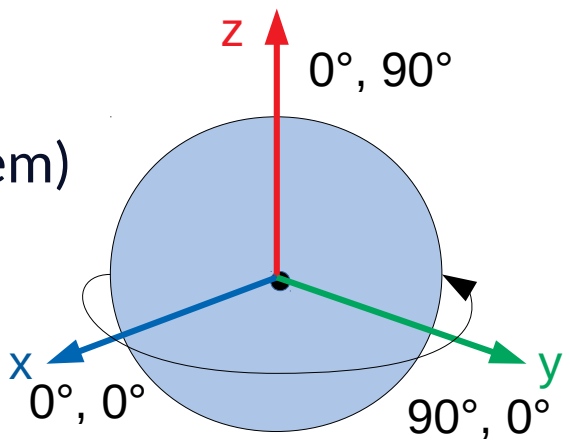
October 10-12 | Berlin, Germany

Generating the Earth Ellipsoid

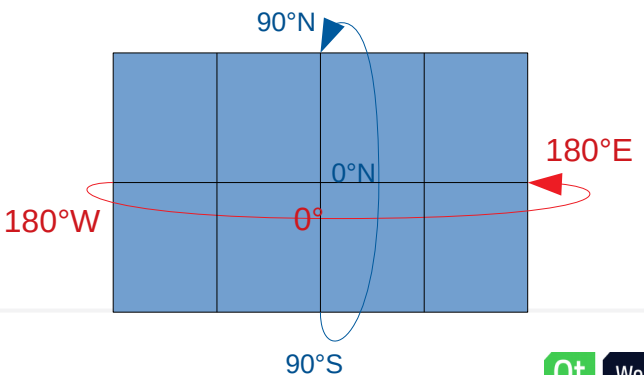
Coordinates Systems



- Geographic Coordinates
  - Latitude, Longitude, Height → Geodetic



- Cartesian Coordinates (using the WGS84 coordinate system)
  - x, y, z relative to the center of the earth
  - Defines earth ellipsoid radii
    (6 378 137km, 6 356 752.3km, 6 378 137km)
  -



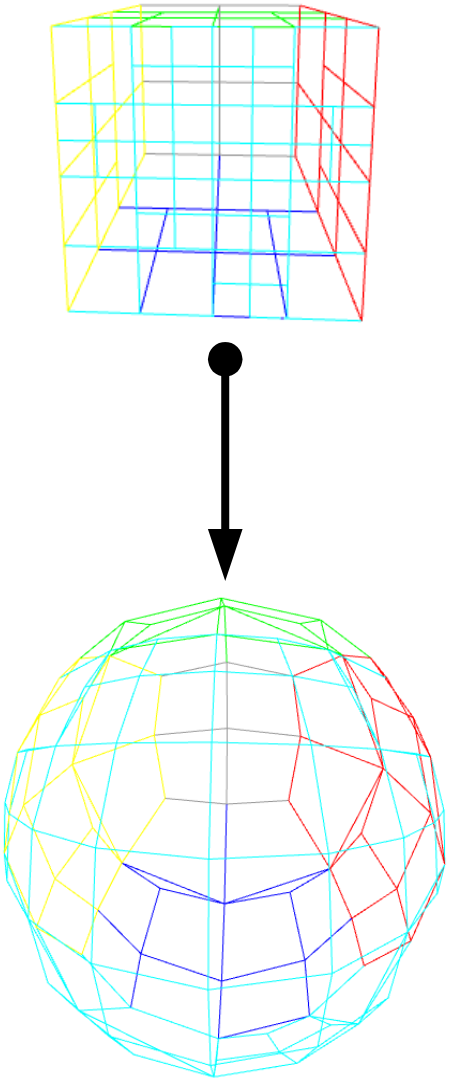
- Projected Coordinates
  - Mercator projection (usually on WGS84 ellipsoid)
    for 2D Maps

Earth Ellipsoid

- Various techniques to create a sphere
  - Geographic Grid Tessellation
    - Result in non uniform tile size but maps easily with imagery
  - Cube Map Tessellation
    - Non uniform tile size but better than the geographic grid, doesn’t map straightaway with imagery
  - Tetrahedron, Octahedron Tessellation
    - Uniform tile size, doesn’t map straightaway with imagery
- We want lots of details, lots of vertices
- But the GPU cannot contain it all at the same time

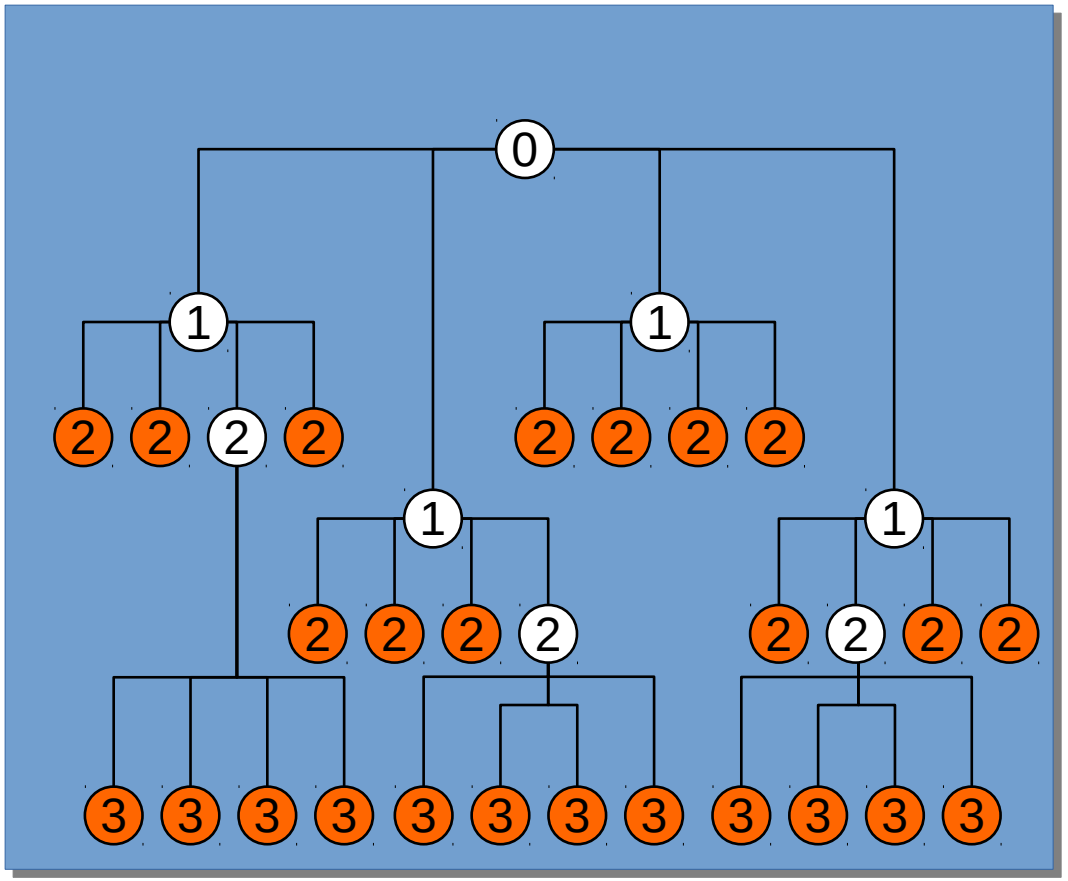
Cube Map Tessellation

- Generate a sphere from a cube
- Subdivision becomes easy
- Cube → Sphere
  - Starting from a unit cube (-1, 1)
  - We normalize each vertex
- Sphere → Ellipsoid
  - We multiply the normalized vertex by the WGS84 radius
    (6 378 137 m, 6 356 752.3 m, 6 378 137 m)
- Area of each subdivision doesn't vary much
  - Not as good as Tetrahedron tessellation though
- Maps well to cubemap texturing



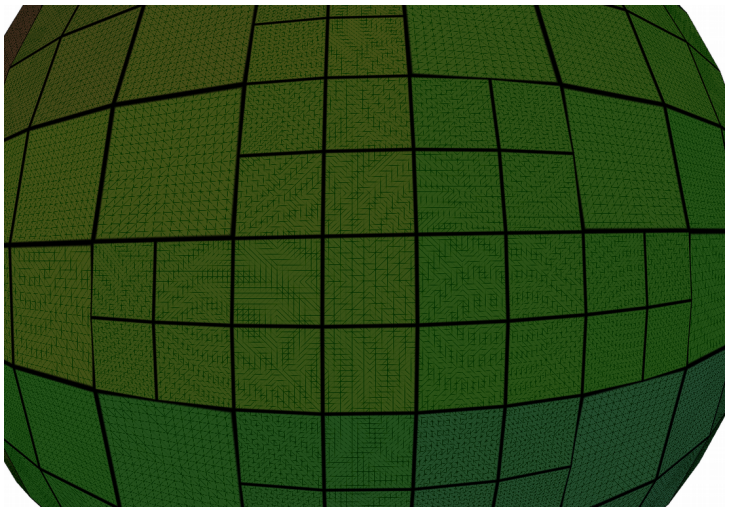
Handling Level of Details

- We want to model earth from space as well as on surface
  - Large Range of Values (km → m)
  - Too much data to be rendered at once
    - Doesn't fit in RAM
    - GPU not powerful enough
    - Not enough disk space
- We need an acceleration structure
  - Divide the world into areas
  - QuadTree, Octree



Adaptative LOD

- We want to subdivide our ellipsoid so that we can have more nodes on the areas we are looking at
  - Our cube/sphere is composed of 6 faces
    - Each face has a root node
    - Each node contains 4 corners
  - For a given node at level n
  - Check if node should be split
    - If yes:
      - subdivide the node into 4 children of level n + 1 and recurse
    - If no:
      - If node is not a leaf, recursively merge it's children
      - If visible, append to the list of nodes to render



When should a node be split?

- When the following conditions are met:
  - When its size is larger than a threshold value
    - Project into screen space and compare size with target tile size (256 x 256)
  - When it's visible

When is a node visible?

- When the following conditions are met:
  - It is a leaf node
  - Not culled by the camera (within the view frustum)
  - Not too far from the camera
    - node's distance to camera < camera's distance to earth center
      - If we assume earth center is (0, 0, 0)
        - |node center – camera| < |(0, 0, 0) -camera|
        - |node center – camera| < |-camera|
  - Has its face normal facing us ($\overrightarrow{viewVector} \circ \overrightarrow{normal} > 0$)

Qt World Summit 2017

October 10-12 | Berlin, Germany

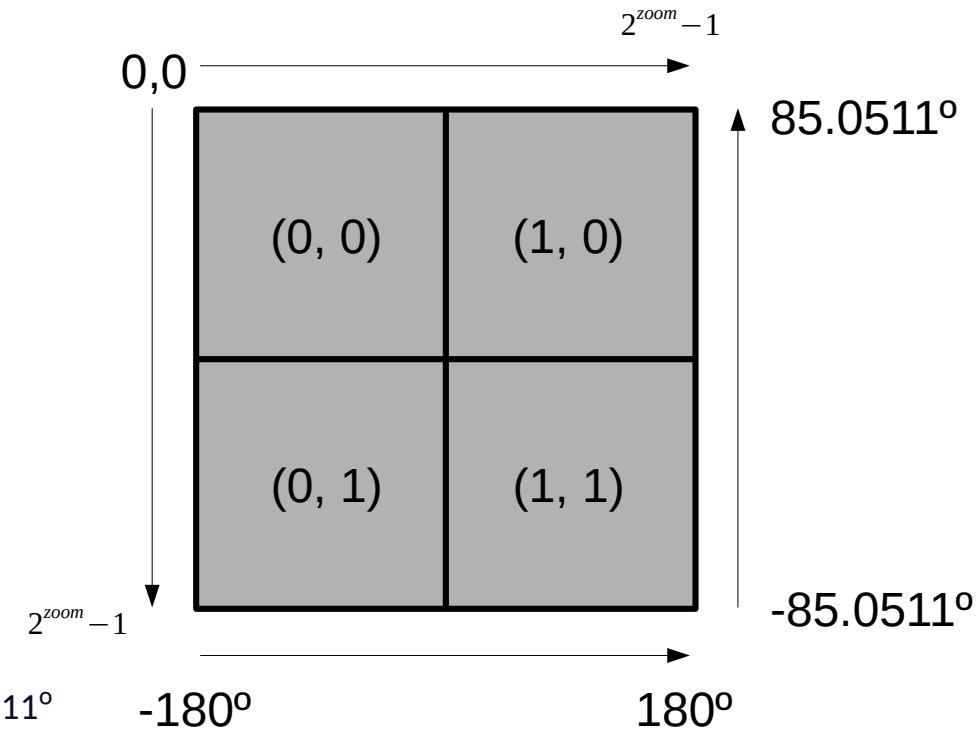
Adding details with imagery tiles

Imagery

- For each geometric tile we need to find the matching imagery tiles
- Various providers
  - Bing Map
  - Google Map
  - Open Street Map
  - ArcGis
  - …
- Various types of layers
  - Satellite
  - Map
  - Heat
  - …

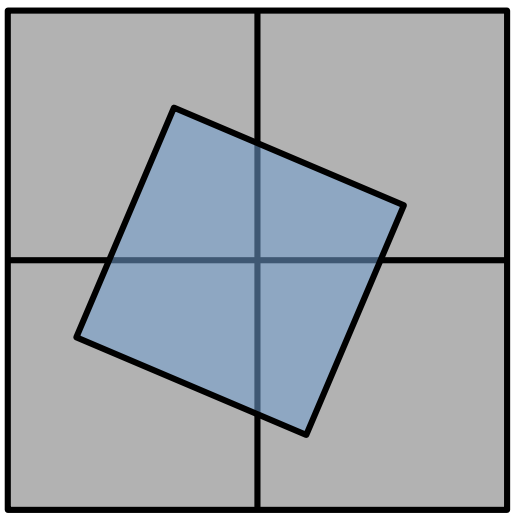
Imagery Layers

- Most imagery layers are provided in Web Mercator projection
- Web Mercator assumes the Earth is a sphere, but we used an Ellipsoid
  - We need to reproject (let's keep that for later)
- Most imagery tiles are accessed using "slippy map" addressing
  - http://someMapLayerProvider/zoom/x/y.png
  - Zoom is usually between 0 and 18
    - At zoom 0, 1 tile covers the whole world
    - At zoom 1, 4 tiles cover the whole world
    - At zoom n, $2^{(2n)}$ tiles cover the whole world
  - X goes from 0 to $2^{zoom}-1 \rightarrow$ 0 is -180°, $2^{zoom}-1$ is 180°
  - Y goes from 0 to $2^{zoom}-1 \rightarrow$ 0 is -85.0511°, $2^{zoom}-1$ is -85.0511°



Mapping Imagery Tiles to Geometric Tiles

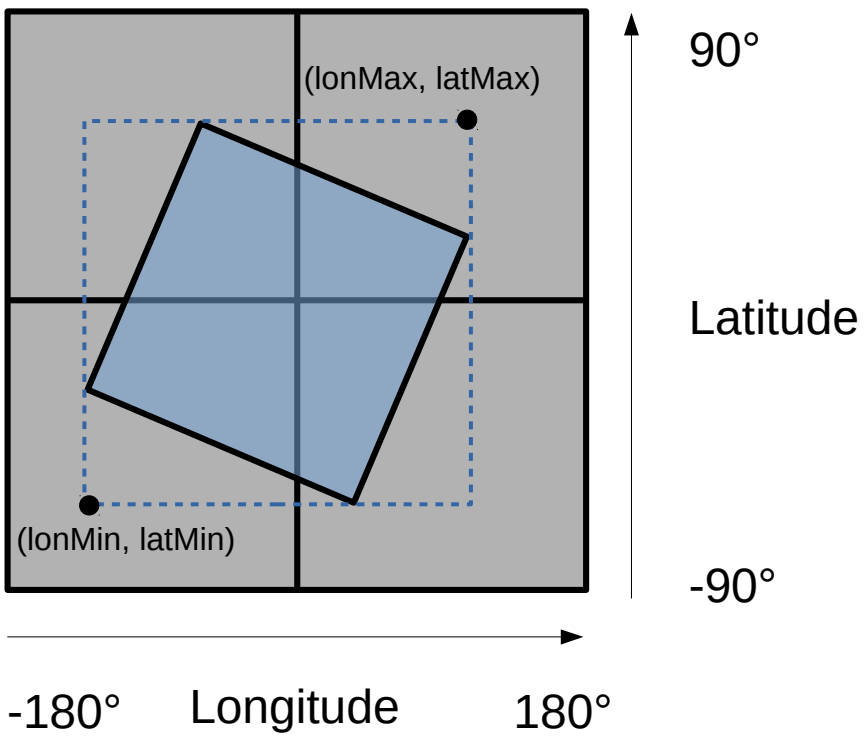
- A geometric tiles may map to several imagery tiles
- For each geometric tile we need to:
  - Find which imagery tiles are required
  - Load the imagery tiles as textures
  - Map these textures on the geometric tile's vertices



Find the Imagery tiles covering a geometric tile

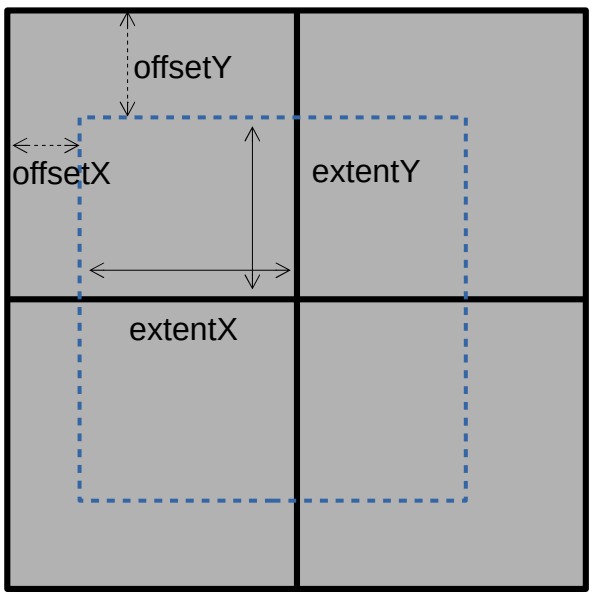
- For each corner of the geometric tile:
  - Retrieve geographic coordinates (lat, lon)
- Retrieve
  - minimum longitude, latitude (lonMin, latMin)
  - maximum longitude latitude (lonMax, latMax)
- Given a zoom level, convert
  - (lonMax, latMax) → toSlippyTileId (xMax, yMax)
  - (lonMin, latMin) → toSlippyTileId (xMin, yMin)
- We can then compute the number of tiles

  (xMax – xMin + 1) * (yMax – yMin + 1)



Mapping Imagery Tiles to Geometric Tiles

- Given the Imagery Tiles required to cover a Geometric Tile
- We then compute the
  - Offset in X and Y
  - The minimum and maximum extent of the tile
  - its scale (node level / tile level)
- At render time, for each geometric tile we provide the GPU with the associated mapping for up to 4 imagery tiles




```
struct TextureInfo
{
        vec4 layerScaleOffsets[4];
        vec4 extentMinMax[4];
};
```

Vector Format Tiles

- Advantages
  - Less data to transmit (binary, json, xml)
  - Finer granularity: select only features you need
- Disadvantages
  - Requires polygon assembly and triangulation
  - More preprocessing overhead
- Rendering:
  - Option 1: Assemble polygons, triangulate and convert to VBO
  - Option 2: Render polygons into Texture
    - Allows to threat these tiles just like Imagery tiles
- Mapzen also provides tiles in vector format

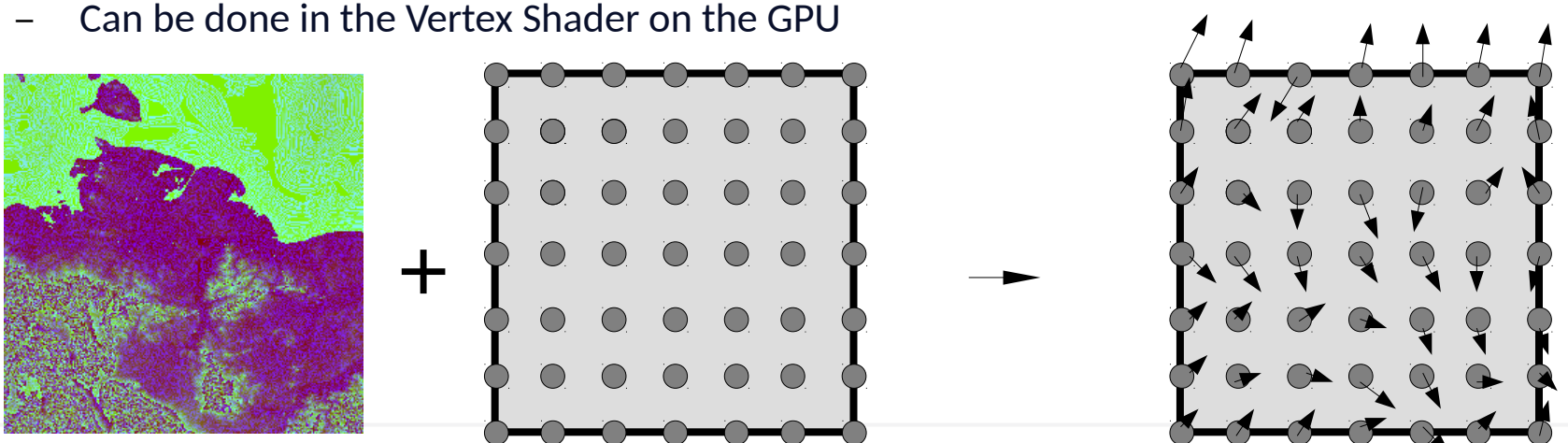
Qt World Summit 2017

October 10-12 | Berlin, Germany

Terrain Elevation

Terrain

- Elevation is provided through elevation textures
  - Mapzen provides elevation tiles
  - www.mapzen.com
- Reuses the same mapping as for imagery tiles
- Elevation tiles are used as heighmaps to displace vertices of geometric terrain tiles
  - Can be done in the Vertex Shader on the GPU

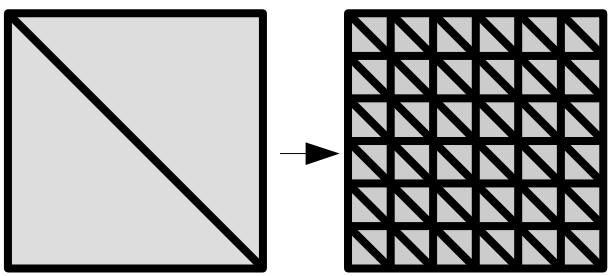


Mapzen Elevation Formats

- 4 formats available
  - Terrarium (.png)
  - Normal (.png)
  - GeoTIFF (.tif)
  - Skadi (.hgt)
- Terrarium
  - PNG tiles
  - 256x256 pixels
  - 32 768 offsets split into red, green blue channels
  - Red and green represent 16 bits of integer
  - Blue represents 8 bits of fraction
  - Height = (red * 256 + green + blue / 256) – 32768
  - Range of elevations span from (-11 000 to 8 900 meters) → (rgb(85, 8, 0) to rgb(162, 198, 0)

Even more details

- We displace the height of each vertex of a geometric tile using the heightmap
  - But wait!!! We only have 4 vertices per tile :(
- We could subdivide more and regroup all the vertices
- Or we could tessellate on the GPU
  - Each geometric tile is initially composed of 4 vertices
  - We can use a tessellation shader to increase that directly on the GPU
    - Each outer edge is subdivided 8 times
    - Each inner edge is subdivided 8 times
    - → 64 quads per tile → 256 vertices



In practice

Drawing

Drawing

- How many draw calls do we need?
  - One per geometric tile?
  - One per imagery tile?
  - Or maybe one draw call is enough?

Drawing

- Option 1
  - 1 draw call to render the whole scene
- Option 2
  - 1) 1 draw call to render the Cube with the textures applied on each face into a CubeMap texture
  - 2) 1 draw call to render the Sphere (looking up textures into the CubeMap is easy)
  - Probably costlier on the GPU than option 1
- Both options have low driver overhead

Building data to be Rendered

- Vertex Data (Geometric Tiles)
  - OpenGL VBO built with Qt3DRender::QBuffer, Qt3DRender::QAttribute
  - For each renderable tile, retrieve its 4 corners and append them a QVector<QVector3D>
    - Perform conversion so that vertex is relative to eye (see vertex precision issues)
    - Set the resulting data array on the Qt3DRender::QBuffer with setData()
- Texture Data (Elevation and Imagery)
  - Texture2DArray with Qt3DRender::QTexture2DArray and Qt3DRender::QTextureImage
    - Allows to group up to 2048 images within a single texture
    - This allows to draw up to 2048 different texture images with a single draw call
- Mapping between Geometric Tiles and Imagery Tiles
  - Uniform Buffer Object
    - Array of structs on GPU memory that can be accessed by shaders
    - Can easily be done by setting a Qt3DRender::QBuffer on a Qt3DRender::QParameter
  - For each tile vertex we associate an index
    - We then use the index to lookup values into the Uniform Buffer Object within our shaders

Precision Issues

- The GPU is good at handling single-precision floats
  - Given the range of values we are working with, single-precision is not enough
- Vertex
  - Causes Jitter
- Depth
  - Causes Z-Fighting

Vertex Precision: Rendering relative to the eye

- Compute world vertex position on the CPU using doubles
- Before uploading the vertices to the GPU
  - Retrieve the eye position E
  - Transform vertex V so that it is relative to the eye: Vrte = V – E
  - Vrte can be converted to a single precision float with a minimal loss of precision
- At render time
  - Transform the ViewMatrix into a viewMatrix relative to the eye (removing the translation vector)

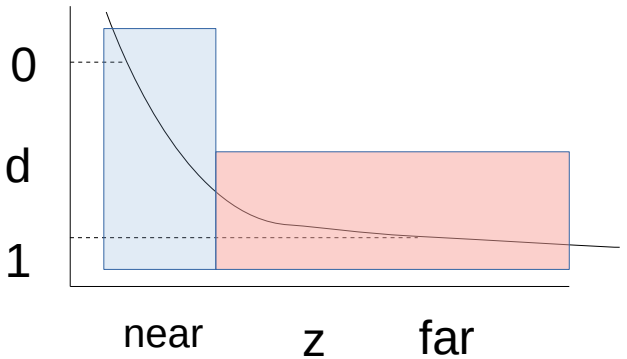
    mat4 viewMatrixRTE = viewMatrix;

    viewMatrixRTE[3] = vec4(0.0, 0.0, 0.0, viewMatrix[3][3]);
  - Perform usual model view projection transformation using viewMatrixRTE

    *gl_Position = projectionMatrix * viewMatrixRTE * vec4(position, 1.0);*

Working around Depth Precision

- We only have 24 bits to store depth values
- Depth is 1 / z
  - As z increases, precision decreases
  - We have less precision for values which are far away
- We can work around that by adapting near and far plane of frustum
- Or we could use a Logarithmic Depth Buffer
  - Allows a better control of the distribution of depth values



Camera Zoom

- Sequence of step
- Each step the distance is half what it was at step n – 1
- Stop when reaching the surface

Code

- Code is available at https://github.com/KDAB/qt3d-examples
- About 1500 LOC of C++ and 500 LOC of QML
- Scripts to download tiles provided
  - You just need to register for a free Mapzen account

Possible Improvements

- Work out texture mapping
- Use a dedicated Qt3D aspect to perform the cube map generation and imagery tile selection
- Perform smarter tessellation on the GPU instead of the fixed number of subdivisions for inner and outer edges
- Fill cracks between tiles
- Use remote loading of textures (should already be possible but not tested)
- Load textures across several frames (right now it loads everything for the frame)
- Better heuristics to predict which tiles to load
- Fine tune tile culling

Suggested Readings

- 3D Engine Design for Virtual Globes by Patrick Cozzi and Kevin Ring
- Rendering the Whole Wide World on the World Wide Web by Kevin Ring
  - https://cesium.com/presentations/files/Rendering%20the%20Whole%20Wide%20World%20on%20the%20World%20Wide%20Web.pptx
- Rendering Massive Terrains using Chunked Level of Detail Control by Thatcher Ulrich
  - http://tulrich.com/geekstuff/sig-notes.pdf
- Depth Precision Visualized by Nathan Reed
  - https://developer.nvidia.com/content/depth-precision-visualized

Thank you