Potree in WebGPU

Markus Schütz, TU Wien, Computer Graphics
https://www.cg.tuwien.ac.at/
About Potree

- WebGL-based rendering of large point clouds
- Up to tens of billions of points
- Below: 18 billion points, (215GB compressed)
The LOD Structure

- Points partitioned into octree
- Each node contains about 1k to 20k points
- Render budget per frame:
  - About 1M to 10M points / 100 to 4000 nodes
Potree in WebGPU

- Depth Buffer Precision
- Compute Shader Rasterization
Depth Buffer Precision
Depth Buffer Precision

- Standard z-buffering has poor precision
  - Hyperbolic distribution & floating point storage
  - All the precision concentrated at near plane
  - Or in OpenGL wherever since \([\text{near, far}] = [-1, 1]\)
- Problematic for:
  - VR: Near needs to be closer than controllers
  - EDL: Shade by analyzing neighboring depth values
Reversed-Z

- WebGPU supports [near, far] of [1, 0] instead of [-1, 1]
- Higher precision at distance by mapping far to 0, near to 1
- Easily integrated by modifying projection matrix
  - [http://austin-eng.com/webgpu-samples/samples/reversedZ](http://austin-eng.com/webgpu-samples/samples/reversedZ)
- EDL: Custom logarithmic depth buffers no longer needed
- VR: Supports close (controllers) and far (100km extent)
- Also: [https://developer.nvidia.com/content/depth-precision-visualized](https://developer.nvidia.com/content/depth-precision-visualized)
Rendering with Compute Shaders
Compute Shader Point Rasterization

- Standard GL_POINTS and "point-list" scale poorly
- Compute shaders 10x faster with >100M points
- AtomicMin to store closest point.

https://github.com/m-schuetz/compute_rasterizer

Basic Approach:

```c
// store closest point
uint64_t point = (depth << 24 | rgb);
atomicMin(framebuffer[pixelID], point);
```

With early-z:

```c
uint64_t oldPoint = framebuffer[pixelID];
uint64_t newPoint = (depth << 24 | rgb);

// store closest point after early-z test
if(newPoint < oldPoint){
    atomicMin(framebuffer[pixelID], newPoint);
}
```
Compute Shader Rendering in WebGPU

- Fastest rasterization requires:
  - 64 bit atomics, subgroups, shuffle, subgroupPartitionNV
- No 64 bit atomics: Render in two 32 bit passes
  - Depth pass, then color pass
- Subgroups in the future?
- Subgroups: Less atomicMins if subgroup writes to same pixel
  - [32 to 1]: subgroupMin: Merge 32 points to 1
  - [2 to 1]: Merge pairs of threads; higher chance to happen
  - [32 to X] subgroupPartitioned: Reduce to affected pixels
Compute Render Performance

- "point-list": Gets slow if lots of points overlap
  - Even worse if sorted by Morton code
- "atomic-min": Fast, even though points rendered 2x
  - Limitation: Only for large VBOs (>1M points/VBO)
  - Potree has too small VBOs (~1k to 10k points / VBO)

<table>
<thead>
<tr>
<th>Method</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-list</td>
<td>66</td>
</tr>
<tr>
<td>Atomic-min</td>
<td>83</td>
</tr>
<tr>
<td>Point-list</td>
<td>27</td>
</tr>
<tr>
<td>Atomic-min</td>
<td>79</td>
</tr>
<tr>
<td>Point-list</td>
<td>7</td>
</tr>
<tr>
<td>Atomic-min</td>
<td>69</td>
</tr>
</tbody>
</table>
LOD Performance
LOD Performance

- Using “point-list” on an RTX 3090
- Using HQS shader (renders points 2x)
- 10 million points, 1054 octree nodes
  - Potree: 54 fps
  - Potree2: 144 fps
- 20 million points
  - Potree: 28 fps
  - Potree2: 98 fps
Future Work

- Adapt LOD structure to make it work with compute rendering
- Compute could also help with "point-list"
  - Generate suitable VBO with compute
  - Render VBO with "point-list"
- Improve compute as new functionality becomes available
  - 64 bit atomics
  - Subgroup communications
  - ...

Thank you for your attention

Potree 2: https://github.com/m-schuetz/Potree2
Demo: https://potree.org/temporary/potree2_ca13/

Compute Rendering (OpenGL): https://github.com/m-schuetz/computeRasterizer
Demo (WebGPU): https://potree.org/temporary/potree2_compute/