Comparison of CPU and GPGPU performance as applied to procedurally generating complex cave systems

Subject: Comp6470 - Special Topics in Computing
Student: Tony Oakden (U4750194)
Supervisor: Dr Eric McCreath
Introduction to Graphic Processing Units

Modern Graphics processing Units contain a great deal of computing power which can be leveraged in two ways:

1. As part of the graphics pipeline

2. Using a general purpose graphics programming API such as OpenCL or Cuda.

GPGPU programming has become a practical choice when solving problems which involve large numbers of calculations.
Objective of this project

• Reduce the time taken to create the mesh data for a procedurally generated cave system

• Investigate the relative benefits of GPGPU and real-time tessellation

• Compare the performance of different hardware configurations

• Better understand *OpenCL* and *Tessellation Shaders*

The Cave creation algorithm

1. Create the cave layout using spheres
2. Create the mesh data - vertices, indexes and normals

The algorithm is particularly attractive because it treats the two stages separately

Procedural Generation of 3D Caves for Games on the GPU, Benjamin Mark, Tobias Mahlmann, Tudor Berechet, Julian Togelius
The Cave creation algorithm

Creates various cave topologies from spherical room components using a pseudo random algorithm.
Mesh creation algorithm

1. Find plane (portals) for each sphere intersection
2. Create mesh data for each sphere
3. Use portals to remove overlapping verts and triangles for each pair of spheres
4. Deform verts using portal data and suitable noise function
5. Weld verts between meshes at joints
Two optimization strategies tested

1. Exploit GPGPU techniques to accelerate the mesh creation process
   - Step 4, the mesh smoothing, is the most computationally intensive
   - The Algorithm lends itself very well to GPGPU approach

2. Produce a lower resolution mesh but put the missing detail back in during the rendering phase using real-time tessellation
Strategy 1: Implementation

• Rewrite mesh smoothing routine in OpenCL
  – Copy the data back to main memory after the GPU has finished
  – The rest of the process, including rendering, remains as it is in the original system
Why use OpenCL?

- Runs on a wide range of hardware and OS
- Looks a lot like C
- Poor development tools
- Relatively easy to get working by following this steps:
  - Convert the C++ routine into C style code
  - Re-organised data into OpenCL friendly forms
  - Test on CPU
  - Move code into OpenCL kernel
  - Watch out for alignment errors
OpenCL results
OpenCL results

Comparison of GPU cards
Strategy 2: Implementation

• Create a lower resolution version of the cave mesh

• Use the *tessellation shader* on the GPU to subdivide polygons and displace the vertices in real-time

*Real-time rendering techniques with Hardware Tessellation – m. Niegbner et al*
Real-time Tessellation – experiment 1

- Implementation is in two parts:
- **Tessellation Control Shader**
  - Controls the level of tessellation and sets up various values for use in the next stage
  - In the first experiment the tessellation level is fixed
  - Second experiment use dynamic tessellation
- **Tessellation Evaluation Shader**
  - Creates the final vertices for the mesh using the data passed through from the TCS
Adding Noise

• When the lower resolution mesh is generated detail is lost – high frequencies alias out so only three octaves of Perlin noise are used.

• Need to add high frequency noise back into the mesh in the tessellation shader: Perlin Noise or Displacement Map

• Calculating the new surface normal is problematic
Comparison of Tessellation Performance

<table>
<thead>
<tr>
<th></th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>307,733</td>
<td>2,498,916</td>
</tr>
<tr>
<td>Triangles</td>
<td>588,514</td>
<td>4,923,817</td>
</tr>
<tr>
<td>Rooms</td>
<td>140</td>
<td>140</td>
</tr>
<tr>
<td>Build time:</td>
<td>8.5</td>
<td>37</td>
</tr>
</tbody>
</table>
GPU Comparison
Conclusion

GPGPU works:
• OpenCL is a good choice for cross platform development
• Test code C style first
• Use CUDA on *Nvidia* devices where speed is important

The case for tessellation is less clear cut:
• Adding extra geometry to meshes is computationally cheaper than tessellation
• Tessellation can add extra value if exploited
• Normals are a problem
Further Work

- Continuing research into the creation of normals
- Conduct a comparison of OpenCL and CUDA
- Refinement of cave algorithm to make more use of GPGPU
- Techniques used for cave mesh creation can be applied to other objects such as: static meshes, vegetation and soft bodied meshes.