Motivation

Want to create, modify and render large geometric models

- Subdivision surfaces
- Gridless simulation techniques
- Particle systems
Motivation

Major bottleneck
- Transfer of geometry to graphics card

Process on GPU if transfer is to be avoided
- Need to avoid intermediate read-back also

Requires dedicated GPU implementations

Perform geometry handling for rendering on GPU
Bus transfer

• Send geometry for every frame
  – because simulation or visualization is time-dependent
  – the user changed some parameter
• Render performance: 12.6 mega points/sec

• Make the geometry reside on the GPU
  – need to create/manipulate/remove vertices without read-back
• Render performance: 114.5 mega points/sec

ATI Radeon 9800Pro, AGP 8x, GL_POINTS with individual color
Motivation

Previous work

- GPU used for large variety of applications
  - local / global illumination [Purcell2003]
  - volume rendering [Kniss2002]
  - image-based rendering [Li2003]
  - numerical simulation [Krüger2003]
- GPU can outperform CPU for both compute-bound and memory-bound applications

Geometry handling on GPU potentially faster
GPU Geometry Processing

Simple copy-existing-code-to-shader solutions will not be efficient

Need to re-invent algorithms, because

- different processing model (stream)
- different key features (memory bandwidth)
- different instruction set (no binary ops)
GPU Geometry Processing

Need shader access to vertex data

• OpenGL SuperBuffer
  – Memory access in fragment shader
  – Directly attach to compliant OpenGL object

• VertexShader 3.0
  – Memory access in vertex shader
  – Use as displacement map

• Both offer similar functionality
OpenGL SuperBuffer

Separate semantic of data from it's storage

• Allocate buffer with a specified size and data layout

• Create OpenGL objects
  – Colors: texture, color array, render target
  – Vectors: vertex array, texcoord array

• If data layout is compatible with semantic, the buffer can be attached to / detached from the object
  – Zero-copy operation in GPU memory
  – Render-to-vertex-array possible by using floating-point textures and render targets
OpenGL SuperBuffer

- Example: floating point array that can be read and written (not at the same time)

```
OpenGL memory object

 RGBA_FLOAT32_ATI

OpenGL texture object

OpenGL render target (offscreen)

change of attachment possible outside rendering activity
```

```cpp
GLint texID = glGenTextures();

glDrawBuffer(GL_FRONT);  // default

GLint buffID = glGenBuffers();

glBindBuffer(GL_TEXTURE_BUFFER, buffID);
glBufferData(GL_TEXTURE_BUFFER, 1024, NULL, GL_DYNAMIC_DRAW);
```

```cpp
void display() {
  glBindTextureUnit(0, texID);
  glBindBuffer(GL_TEXTURE_BUFFER, buffID);
  glDrawBuffer(GL_FRONT);  // default

  glBindBuffer(GL_TEXTURE_BUFFER, 0);
  glBindTextureUnit(0, 0);

  glBindBuffer(GL_TEXTURE_BUFFER, buffID);
  glDrawBuffer(GL_FRONT);  // default

  glBindBuffer(GL_TEXTURE_BUFFER, 0);
  glBindTextureUnit(0, 0);
}
```
GPU Particle Engine

Demo
Overview

GPU particle engine features

• Particle advection
  – Motion according to external forces and 3D force field
• Sorting
  – Depth-test and transparent rendering
  – Spatial relations for collision detection
• Rendering
  – Individually colored points
  – Point sprites
Particle Advection

Simple two-pass method using two vertex arrays in double-buffer mode

- Render quad covering entire buffer
- Apply forces in fragment shader
Sorting

Required for correct transparency and collision detection

- Bitonic merge sort (sorting network) [Batcher1968]
- Sorting n items needs (log n) stages
- Overall number of passes $\frac{1}{2} (\log^2 n + \log n)$
Sorting a 2D field

- Merge rows to get a completely sorted field

- Implement in fragment shader [Purcell2003]
  - A lot of arithmetic necessary
  - Binary operations not available in shader
Fast sorting

Make use of all GPU resources

• Calculate constant and linear varying values in vertex shader and let raster engine interpolate

• Render quad size according to compare distance

• Modify compare operation and distance by multiplying with interpolated value
Fast sorting

- Perform mass operations (texture fetches) in fragment shader

- $t_0 = \text{fragment position}$
  - $t_1 = \text{parameters from vertex shader (interpolated)}$

```plaintext
OP1 = TEX[t0]

sign = (t1.x < 0) ? -1 : 1

OP2 = TEX[t0.x + sign * dx, t0.y]

return (OP1 * t1.y < OP2 * t1.y) ? OP1 : OP2
```
Fast sorting

- Final optimization: sort [index, key] pairs
  - pack 2 pairs into one fragment
  - lowest sorting pass runs internal in fragment shader
- Generate keys according to distance to viewer or use cell identifier of space partitioning scheme
Fast sorting

- Same approach for column sort, just rotate the quads
- Benefits for full sort of n items
  - $2 \cdot \log(n)$ less passes (because of collapse and packing)
  - $n/2$ fragments processed each pass (because of packing)
  - Workload balanced between vertex and fragment shaders (because of rendering quads)

→ Speedup factor of 10 compared to previous solutions
## Fast sorting

- **Performance: full sort**

<table>
<thead>
<tr>
<th>$n$</th>
<th>sorts/sec</th>
<th>mega items/sec</th>
<th>mega frag/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128^2$</td>
<td>175.0</td>
<td>2.8</td>
<td>130</td>
</tr>
<tr>
<td>$256^2$</td>
<td>43.6</td>
<td>2.8</td>
<td>171</td>
</tr>
<tr>
<td>$512^2$</td>
<td>9.3</td>
<td>2.4</td>
<td>186</td>
</tr>
<tr>
<td>$1024^2$</td>
<td>1.94</td>
<td>2.0</td>
<td>193</td>
</tr>
<tr>
<td>$128^2$</td>
<td>238.0</td>
<td>3.9</td>
<td>177</td>
</tr>
<tr>
<td>$256^2$</td>
<td>109.0</td>
<td>7.1</td>
<td>429</td>
</tr>
<tr>
<td>$512^2$</td>
<td>24.4</td>
<td>6.4</td>
<td>489</td>
</tr>
<tr>
<td>$1024^2$</td>
<td>4.85</td>
<td>5.1</td>
<td>483</td>
</tr>
</tbody>
</table>

- ATI Radeon 9800Pro
- ATI Radeon X800 XT
Particle – Scene Collision

Additional buffers for state-full particles

- Store velocity per particle (Euler integration)
- Keep last two positions (Verlet integration)
- Simple: Collision with height-field stored as 2D texture
  - RGB = [x,y,z] surface normal
  - A = [w] height
  - Compute reflection vector
  - Force particle to field height
Particle – Particle Collision

Essential for natural behavior

- Full search is $O(n^2)$, not practicable
- Approximate solution by considering only neighbors
- Sort particles into spatial structure
  - Staggered grid misses only few combinations
 Particle – Particle Collision

- Check \( m \) neighbors to the left/right
- Collision resolution with first collider (time sequential)
- Only if velocity is not excessively larger than integration step size

solve quadratic equation on GPU
GPU Particle Engine

• Acknowledgements

ATI Research for providing hardware

Jens Krüger for his help on shader programming

http://www.cg.in.tum.de/GPU