Hardware-Accelerated Ray-Triangle Intersection Testing for High-Performance Collision Detection

Sung-Soo Kim, Seung-Woo Nam, Do-Hyung Kim and In-Ho Lee

Electronics and Telecommunications Research Institute
Goal

- Perform a fast ray-triangle intersection computation of massive models.
- Design and implement a novel FPGA-accelerated architecture for fast collision detection among rigid bodies.
  - Support 13 intersection types among rigid bodies.
  - FPGA-accelerated implementation for accelerating intersection computations among collision primitives.
Motivation

- Fast rendering for massive models and complex scenes
Problem

- **Collision Query**
  - checks whether two objects intersect and returns all pairs of overlapping features.

- **Real-time collision queries**
  - remain one of the major bottlenecks for interactive physically-based simulation and ray tracing.

- **Key Challenge**
  - to develop the custom hardware for collision detection and ray tracing
Ray-Triangle Intersection of Massive Models
Main Contributions

- Direct applicability to collision objects with dynamically changing topologies
- Sufficient memory to buffer the ray intersection input and output data
- Up to an order of magnitude faster runtime performance over prior techniques for ray-triangle intersection testing
- Interactive collision query computation on massive models.
Related Work

Collision Detection
- BVHs (sphere tree, OBB-tree, AABB-tree, k-DOP-tree), octree and k-d tree
- overhead for each time interval tested, spent updating bounding volumes and collision pruning data structures

Programmable GPU
- a general purpose SI MD processor
- GPU-based ray tracing approaches
- GPU cannot gain a significant speed-up over a pure CPU-based implementation.

Custom Hardware
- AR350 processor
- RPU, DRPU
Hardware Architecture

Input registers
- Counter
- Primary
- Secondary
- Transformer

Collision detection engine
- Acceleration structures
- Primitive intersection testing

function selector
ready

Output registers
- Collision position
- Collision flag
- Distance (T) value

Update Engine

Memory controller

Buffers
- Collision position
- F-Index
- Stencil-T
Custom Hardware for Collision Detection

**Specifications**
- 64bits/66MHz PCI interface.
- PCI Controller: Xilinx V2P20
- Collision Detection Engine: Xilinx V2P70
- Two 1GB DDR memories (288 bus input bus)
- Seven 2MB SRAMs (224 bit output bus)
Hardware-Accelerated Ray Triangle Intersection Testing

1: procedure HW-AcceleratedRayTriangleIntersection
2: input: \( \mathcal{P}, S \)
3: output: \( \mathcal{R} \) (CP, F-value, index, T-value)
4: collisionType CT = RAY_TRIANGLE;
5: initializeDevice();
6: secondaryUpload(S);
7: for \( \forall O_k, D_k \in \mathcal{P} \) do
8: primaryRegFileUpload(O_k, D_k);
9: invokeCDE(CT);
10: \( \mathcal{R} \leftarrow \) downloadSRAM();
11: return \( \mathcal{R} \)

Algorithm 1: Hardware-Accelerated Ray Triangle Intersection Testing.
Hardware-Accelerated Ray Triangle Intersection Testing

1: procedure HW-AcceleratedRayTriangleIntersection
2: input: \( \mathcal{P}, S \)
3: output: \( \mathcal{R} \) (CP, F-value, index, T-value)
4: collisionType CT = RAY_TRIANGLE;
5: initializeDevice();
6: secondaryUpload(S);
7: for \( \forall O_k, D_k \in \mathcal{P} \) do
8: primaryRegFileUpload(\( O_k, D_k \));
9: invokeCDE(CT);
10: \( \mathcal{R} \leftarrow \) downloadSRAM();
11: return \( \mathcal{R} \)

Algorithm 1: Hardware-Accelerated Ray Triangle Intersection Testing.
Hardware-Accelerated Ray Triangle Intersection Testing

Algorithm 1: Hardware-Accelerated Ray Triangle Intersection Testing.

1: procedure HW-AcceleratedRayTriangleIntersection
2:       input : $\mathcal{P}, S$
3:       output : $\mathcal{R}$ (CP, F-value, index, T-value)
4:       collisionType CT = RAY_TRIANGLE;
5:       initializeDevice();
6:       secondaryUpload($S$);
7:       for $\forall O_k, D_k \in \mathcal{P}$ do
8:           primaryRegFileUpload($O_k, D_k$);
9:           invokeCDE(CT);
10:      $\mathcal{R} \leftarrow$ downloadSRAM();
11:      return $\mathcal{R}$
Algorithm 1: Hardware-Accelerated Ray Triangle Intersection Testing.

1: procedure HW-AcceleratedRayTriangleIntersection
2: input : \( \mathcal{P}, S \)
3: output : \( \mathcal{R} \) (CP, F-value, index, T-value)
4: collisionType CT = RAY_TRIANGLE;
5: initializeDevice();
6: secondaryUpload(S);
7: for \( \forall O_k, D_k \in \mathcal{P} \) do
8: \hspace{1cm} primaryRegFileUpload(O_k, D_k);
9: \hspace{1cm} invokeCDE(CT);
10: \hspace{1cm} \mathcal{R} \leftarrow downloadSRAM();
11: \hspace{1cm} return \( \mathcal{R} \)
Algorithm 1: Hardware-Accelerated Ray Triangle Intersection Testing.

1: procedure HW-AcceleratedRayTriangleIntersection
2: input : $\mathcal{P}, \mathcal{S}$
3: output : $\mathcal{R}$ (CP, F-value, index, T-value)
4: collisionType CT = RAY_TRIANGLE;
5: initializeDevice();
6: secondaryUpload($\mathcal{S}$);
7: for $\forall O_k, D_k \in \mathcal{P}$ do
8:    primaryRegFileUpload($O_k, D_k$);
9:    invokeCDE(CT);
10:   $\mathcal{R} \leftarrow$ downloadSRAM();
11: return $\mathcal{R}$
Algorithm 1: Hardware-Accelerated Ray Triangle Intersection Testing.

1: procedure HW-AcceleratedRayTriangleIntersection
2: input : $\mathcal{P}, S$
3: output : $\mathcal{R}$ (CP, F-value, index, T-value)
4: collisionType CT = RAY_TRIANGLE;
5: initializeDevice();
6: secondaryUpload($S$);
7: for $\forall O_k, D_k \in \mathcal{P}$ do
8: primaryRegFileUpload($O_k, D_k$);
9: invokeCDE(CT);
10: $\mathcal{R} \leftarrow$ downloadSRAM();
11: return $\mathcal{R}$
Collision Detection Engine

- A modular hardware component for performing the collision computations.
- consists of acceleration structures and primitive intersection testing components.
- 13 types of intersection queries
  - Ray-triangle, OBB-OBB, triangle-OBB, triangle-OBB, sphere-sphere, triangle-sphere, ray-cylinder, triangle-cylinder, cylinder-cylinder, OBB-cylinder, OBB-plane, ray-sphere, and sphere-OBB
- Pipelined technique for increasing instruction throughput
- Four outputs
  - collision flag, collision position, index, separation distance or penetration depth
Update Engine

- Simplify routing lines in the hardware
- Make memory controller efficient by coupling buffers
  - F-index buffer
  - 2 stencil buffers
- Single precision floating point of IEEE standard 754
Analysis of Intersection Algorithms

Three ray triangle intersection algorithms
- Badouel's algorithm
- Möller and Trumbore's algorithm
- the algorithm using Plücker coordinates

Algorithm comparison in terms of the latency, the number of I/O and hardware resources

Möller's algorithm has been more efficient than others in view of the processing speed and usage of storage.
# Analysis of Intersection Algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th># of inputs</th>
<th># of outputs</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Badouel’s</td>
<td>9</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>Möller’s</td>
<td>9</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Plücker’s</td>
<td>15</td>
<td>6</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 1: Comparison of ray-triangle intersection algorithms in terms of the number of inputs, the number of outputs and latency for hardware implementation.
## Analysis of Intersection Algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Badouel’s</th>
<th>Möller’s</th>
<th>Plücker’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>27</td>
<td>27</td>
<td>54</td>
</tr>
<tr>
<td>Divider</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Adder</td>
<td>13</td>
<td>12</td>
<td>31</td>
</tr>
<tr>
<td>Subtractor</td>
<td>23</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>Comparator</td>
<td>6</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>AND</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Analysis of the hardware resource for ray-triangle intersection algorithms.
Implementation

- Intel Xeon 2.0GHz (2GB memory)
- NVIDIA GeoForce 7800GT GPU
- C++/OpenGL/Cg
- VHDL implementation
  - Xilinx ISE, ModelSim
Comparison

- Three configurations of collision detections
  - Static objects vs. static objects
  - Static objects vs. dynamic objects
  - Dynamic objects vs. dynamic objects

Test terrain: 259,572 triangles
Static objects vs. static objects

Figure 5: The comparison result of the ray-triangle intersection testing (static objects vs. static objects).
Static objects vs. dynamic objects

Figure 6: The comparison result of the ray-triangle intersection testing (static objects vs. dynamic objects).
Dynamic objects vs. dynamic objects

Figure 7: The comparison result according to the number of objects.
Analysis and Limitations

Benefits

- Data reusability
  - transformer to avoid the re-transmission bottleneck
- Runtime performance
  - instruction pipelining to improve the throughput of the collision detection engine

Limitations

- We could not implement the acceleration structures in our hardware architecture.
- If traversal of acceleration structures is performed in CPU, we can improve the performance.
Conclusion

- Novel dedicated hardware architecture to perform collision queries.
  - Ray-triangle intersection
  - Sphere-sphere intersection

- The proposed hardware-accelerated approach could prove to be faster than
  - CPU-based algorithm: 70x improvement
  - GPU-based algorithm: 4x improvement

- Future work
  - Hardware-acceleration structures for dynamic scenes.
Thank you!

Questions?

e-mail: sungsoo@etri.re.kr