Object-Space Interference Detection on Programmable Graphics Hardware

Alexander Greß and Gabriel Zachmann
University of Bonn

Motivation
- Collision detection is a fundamental task in
  - Virtual Prototyping
  - Haptic rendering (force-feedback)
  - Physically-based simulation (rigid bodies etc.)
  - Medical training/planning systems
- Collision detection performance is critical for
  - Responsive VR systems
  - Real-time simulation
  - Natural interaction
→ Need of hardware accelerated algorithms

Previous Work
- Collision detection in graphics hardware:
  - image-space algorithms:
    - RECODE [Baciu et al. 1999]
    - CInDeR [Knott, Pai 2003]
    - CULLIDE [Govindaraju et al. 2003]
    and further image-space methods
  → restricted to objects of certain shape and connectivity
- Hierarchical collision detection
  - OBBs [Gottschalk et al. 1996]
  - DOPs, AABBs [Zachmann 1998, 2002]
  - Convex surface decomposition [Ehmann et al. 2001]

Programmable Graphics Hardware (GPU)
- parallel architecture of GPU:
  - multiple vertex program / fragment program execution units
  - vertex and fragment programs are designed to run with an arbitrary number of execution units
  → scalability to future GPUs
- all calculations in floating point
  (up to 32 bits precision)
- SIMD instruction set
→ high floating point throughput
GPU architecture overview

Our Goal
- Collision detection on current graphics hardware
  - using programmable graphics hardware (GPU)
  - utilizing its SIMD capabilities and high floating point throughput (using floating point textures for storage)
  - implementing an hierarchical algorithm
  - exact interference detection in object-space
  - no requirements on shape, topology, connectivity

Bounding Volume Tree
- inner nodes: bounding volumes (AABBS in our approach)
- leaf nodes: triangles
- Simultaneous traversal of two trees:
  - all pairs of nodes \((S_i, T_j)\) are considered, where \(S_i\) is a node of tree \(S\) and \(T_j\) is a node of tree \(T\) on the same hierarchy level
  - for a pair of inner nodes \((S_i, T_j)\) their child nodes have to be checked only if the bounding volumes (BVs) corresponding to \(S_i\) and \(T_j\) overlap
- Our traversal scheme:
  - breadth-first strategy (to exploit parallelism)

Simultaneous overlap testing of multiple BVs
- Central task of the breadth-first traversal:
  - given: list \(L\), tree node \(T\)
  - determine: list of those nodes from \(L\) that overlap with \(T\)
- Pseudocode:

```
overlappingChildren(list L, node T): list
  for all nodes S from list L do
    for all children S_i of S do
      if S_i and T overlap then
        L'.append(S_i);
  return L';
```
Simultaneous overlap testing of multiple BVs

Idea: implement as fragment program

- theoretically, all overlap tests could be executed in parallel as they are independent of each other
- parallel execution requires a data structure that allows direct access to elements (arrays); lists are unsuitable
- arrays can be represented on the graphics hardware by (floating-point) textures

→ make loop vectorizable by using arrays instead of lists

Simultaneous overlap testing of multiple BVs

Naïve approach: use arrays with NULL-elements

overlappingChildren \( (\text{array } a, \text{node } T) \): array

array \( a' \):

for all nodes \( S_j \) from array \( a \) do
  for all children \( S_{ij} \) of \( S_j \) do
    if \( S_{ij} \) and \( T \) overlap then
      \( a'[2j+i] := S_{ij}; \)
    else
      \( a'[2j+i] := \text{NULL}; \)

return \( a' \);

→ vectorizable, but unsuitable for parallel execution by a fragment program where one execution unit is assigned for each output array element

Solution: tightly-packed arrays

1. Calculate overlap counts for the children of all nodes contained in the input array (i.e. 1 if there is an overlap, 0 otherwise)

2. Build a tree by summing up overlap counts corresponds to a \( \text{mip-map} \); total size \( O(n) \)
Simultaneous overlap testing of multiple BVs

3. Successively construct the output array

output:

0 1 1 1 0 1 1 0 1
The overall simultaneous traversal scheme

Pseudocode using a queue:

```
traverse (node S, node T):
queue q;
array a := { S };
q.insert(a, T);
while q is not empty do
    (a, T) := q.top;
    q.pop;
    for all children T_i of T do
        array a' := overlappingChildren(a, T_i);
        q.append(a', T_i);
```

The overall simultaneous traversal scheme

Subroutine `overlappingChildPairs()`:
- is vectorizable as an array is used for input/output and there are no other dependencies between iterations
- its subroutine `overlappingChildren()` is – as described – executed by a fragment program

Idea: implement as vertex program
- the input array can be specified using vertex array(s)
- the output array must be written to vertex array(s), too

requires the new `ARB_super_buffer` OpenGL extension

The overall simultaneous traversal scheme

Pseudocode using 2D arrays:

```
traverse (node S, node T):
array a := (S);
array b := (a, T);
while b is not empty do
    b := overlappingChildPairs(b);

overlappingChildPairs (array b): array b*:
    for all (a, T_j) from array b do
        for all children T_{ij} of T_j do
            array a'_ij := overlappingChildren(a, T_{ij});
            b'[2j+i] := (a'_ij, T_{ij});
```

Implementation details

Mapping of data structures to GPU memory:
- one call of `overlappingChildPairs()` corresponds to rendering \( n \) lines of lengths \( m_0 \ldots m_{n-1} \) into a 2D buffer, where \( n \) is the length of array \( b \) and \( m_i \) is the length of array \( a_i \)
- the nodes of tree \( S \), which are referenced by the elements of arrays \( a_i \), are stored in sets of 1D textures (up to three textures per hierarchy level)
- the nodes of tree \( T \), which are referenced by the elements of array \( b \), are stored in vertex arrays (one per hierarchy level)
- the lengths of the arrays \( a_i \), which are determined inside the subroutine `overlappingChildren()`, are written to an additional vertex array (using `ARB_super_buffer` extension)
- transformation matrices for trees \( S \) and \( T \) can be passed to the fragment and vertex program units as program parameters
Implementation details

- Hardware limitations:
  - the number of nodes for each hierarchy level (and therefore the number of triangles of a single mesh) may not be larger than the max. allowed texture size $M$ (usually $M=2048$)
  - larger meshes have to be split into multiple sub-meshes with max. $M$ triangles each

- Possible optimizations:
  - avoid unnecessary calls of `overlappingChildPairs()` when array $b$ contains only empty arrays $a_i$ (can be determined by querying an occlusion count using the ARB_occlusion_query extension)
  - by using 2D textures of height $M$ for every hierarchy level $i$ and packing multiple 2D arrays into these textures, $M/2$ meshes can be processed simultaneously by a single batch (i.e. a single `overlappingChildPairs()` call)

Conclusions and Future Work

- Summary:
  - hierarchical collision detection using programmable graphics hardware
  - all calculations done in object-space, not image-space
  - no requirements on shape, topology, connectivity

- Ongoing and future work:
  - in-depth performance analysis of our implementation
  - the usage of bounding volumes other than AABBs and of enhanced tree traversal schemes are to be evaluated

Questions?