TRaX programming examples

Using global memory

Non-recursive tree traversal
TRaX programming recap

• Local memory (private to every thread)
  – Handled by compiler (stack space)
  – You will never need to explicitly deal with it

• Global memory
  – Pre-loaded by simulator
  – Explicitly accessed
  – loadf, storef, loadi, storei
Global Scene Data

- Light
- Camera
- Model (BVH/grid and triangles)
- Materials
Recap

- First few words of memory hold constants and pointers (LoadMemory.cc):
  1: width
  2: 1 / width
  3: float width
  4: height
  5: 1 / height
  6: float height
  7: start_fb
  8: start_scene
  9: start_matls
  10: start_camera
  11: background
  12: start_light
  14: end_memory
  15: size - 1
  16: ray depth
  17: num samples
  18: epsilon
  28: start_triangles
  29: num_triangles
simhwrt arguments

- Many of these values can be set with arguments to the simulator

- `--view-file` (loads camera data)
- `--model` (loads the BVH and materials for you)
- `--light-file` (loads the light position)
- `--config-file` (configures the TRaX HW)
- `--epsilon`

- `./simhwrt --help` to see all options
simhwrt arguments

• More useful options:
  • --num-thread-procs  (number of threads per TM)
  • --num-cores        (number of TMs)
  • --num-l2s          (number of TM clusters)
  • --no-cpi           (don’t print run stats)
  • --issue-verbosity  (set to 1 for per-cycle details)
--write-dot

- --write-dot <depth>

- Generates “bvh.dot” graph representation of the BVH

- Use the dot tool:
  - dot –Tgif bvh.dot –o bvh.gif
  - Part of the graphviz package

- Can be useful for debugging BVH traversal
• Restricted to a depth of 4
Loading the camera

- \texttt{start\_camera} = \texttt{loadi(0, 10)}
- \texttt{eye[x, y, z]} is at \texttt{start\_camera + [0..2]}
- \texttt{corner} : \texttt{start\_camera + [3..5]}
- \texttt{across} : \texttt{start\_camera + [6..8]}
- \texttt{up} : \texttt{start\_camera + [9..11]}
- \texttt{gaze} : \texttt{start\_camera + [12..14]}
- \texttt{u} : \texttt{start\_camera + [15..17]}
- \texttt{v} : \texttt{start\_camera + [18..20]}
Loading the camera

Vector eye( loadf( start_camera, 0 ),
            loadf( start_camera, 1 ),
            loadf( start_camera, 2 ) );

Vector up( loadf( start_camera, 9 ),
           loadf( start_camera, 10 ),
           loadf( start_camera, 11 ) );

Vector gaze( loadf( start_camera, 12 ),
             loadf( start_camera, 13 ),
             loadf( start_camera, 14 ) );
Loading the camera

- Recommend a constructor which takes an address

```cpp
PinholeCamera::PinholeCamera(int addr) {
    eye = loadVectorFromMemory(addr);
    up = loadVectorFromMemory(addr + 9);
    lookdir = loadVectorFromMemory(addr + 12);
    u = loadVectorFromMemory(addr + 15);
    v = loadVectorFromMemory(addr + 18);
}

PinholeCamera camera(loadi(0, 10));
```
inline Vector loadVectorFromMemory(const int &address)
{
    float x, y, z;
    x = loadf(address, 0);
    y = loadf(address, 1);
    z = loadf(address, 2);

    return Vector(x, y, z);
}

The light doesn’t specify a color (assume white)

```c
inline PointLight loadLightFromMemory(int addr)
{
    return PointLight(loadVectorFromMemory(addr), Color(1.f, 1.f, 1.f));
}

PointLight light = loadLightFromMemory(loadi(0, 12));
```
Triangles

- Triangles are stored as 11 words:
  - p1[x, y, z] (address + 0..2)
  - p2[x, y, z] (address + 3..5)
  - p3[x, y, z] (address + 6..8)
  - ID (address + 9)
  - material ID (address + 10)
Triangles

Vector e1( loadf(addr, 0 ),
    loadf(addr, 1 ),
    loadf(addr, 2 ) );
Vector e2( loadf(addr, 3 ),
    loadf(addr, 4 ),
    loadf(addr, 5 ) );
Vector e3( loadf(addr, 6 ),
    loadf(addr, 7 ),
    loadf(addr, 8 ) );

• Encapsulate this in a helper (constructor, etc)
• Don’t call your class “Triangle”!
inline Vector normal() const
{
    Vector edge1 = p1 - p3;
    Vector edge2 = p2 - p3;
    Vector n = Cross(edge1, edge2);
    n.normalize();
    return n;
}

• Don’t compute normals unless you need to shade that triangle
Try to avoid unnecessary memory traffic

Don’t load material ID every time a triangle is tested for intersection

Save the address of the closest hit triangle (hitRecord)

Then only perform load of material ID once during shading
Traversing the scene

• Before we get in to BVH traversal, a simpler example

• use start_triangles, and num_triangles

• Simply loop through every triangle, loading them from memory
Traversing the scene

```cpp
int start_tris = loadi(0, 28);
int num_tris = loadi(0, 29);
for(int i=0; i < num_tris; i++)
{
    Tri t =
        loadTriFromMemory(start_tris + (i * 11));
    t.intersect(hitRec, ray);
}
```
The BVH is laid out in memory as follows:

- `c_min` and `c_max` (2 box corners, 6 floats)
- `child ID`
- `num children`
- `start_bvh`
- `start_bvh + 8`

A single BVH node contains:

- `c_min` and `c_max` (2 box corners, 6 floats)
- `1` for the child count
- `-1` for the null child
- `3` for the number of children
- `-1` for the null child

This layout allows for efficient memory access and evaluation in rendering tasks.
BVH layout

• Sibling nodes are next to each other in memory

• Right child’s ID is always left_id + 1
BVH layout
Traversing the BVH

• We don’t want to use recursion
  – Stack frames will quickly outgrow the local memory space
  – Inline function calls are faster

• But we need to traverse a tree (inherently recursive)

• Use a software-managed stack

• int stack[32]; // holds node IDs
• int sp = 0; // stack pointer
Pseudo code

current_node = root
while(true)
    if(ray intersects current node)
        if(interior node)
            push right child
            current = left child
            continue;
        else
            intersect all triangles in leaf
    if(stack is empty)
        break;
break;
current = pop stack
Example

```c
inline void intersect(HitRecord& hit,
const Ray& ray) const {
    int stack[32];
    int node_id = 0;
    int sp = 0;
    while(true){
        int node_addr = start_bvh + node_id * 8;
        Box b = loadBoxFromMemory(node_addr);
        HitRecord boxHit;
        b.intersect(boxHit, ray);
        if(boxHit.didHit())
            // and so on...
    }
```

left_id = loadi( node_addr, 7 );
int num_children = loadi( node_addr, 6 );
if ( num_children < 0 )
{
    stack[ sp++ ] = left_id + 1;
    continue;
}

tri_addr = left_id;
for ( int i = 0; i < num_children; ++i)
// ...
Implementation

```cpp
inline void intersect(HitRecord& hit,
                      const Ray& ray) const
{
    
    • Note that this hit record passed in is for the final hit triangle (or none if background)

    • Don’t use the same one for testing against boxes!

    • Store the address of the closest triangle in `hit` (used later for shading)
```
Implementation

for each pixel...
    Ray ray;
    camera.makeRay(ray, x, y);
    HitRecord hit;
    bvh.intersect(hit, ray);
    result = shade(hit, ray, bvh, light, start_matls);
Important notes

• Remember, the BVH is in global memory
• Don’t try to rebuild it in local memory

• My bvh class contains just a pointer to start_scene

    BoundingVolumeHierarchy(const int & _start_scene)
    {
        start_bvh = _start_scene;
    }

• Nodes are loaded 1 at a time as needed
Important notes

• Remember that for leaf nodes, child pointer is an absolute address

• Address of the first triangle
Performance

- Remember, there are some optimizations:
  - Traverse down closer child first
  - Don’t traverse subtree if closer triangle already found
  - The pseudo-code I’ve shown doesn’t do this
Programs 3, 4

• Both will be available for those who want to skip ahead

• Program 3:
  – Render Cornell scene by looping through triangles
  – Render un-shaded box (for verification of correct ray-box test)

• Program 4:
  – Render Cornell scene using BVH
  – Render conference scene (would never finish without BVH)
Program 3

Un-shaded box to verify correct ray-box intersect
Program 3

Rays originating inside the box are often a source of trouble.

Most rays will originate inside BVH.
Program 4

We will give you plenty of other models to play with as well
Box normals

\[
\begin{align*}
\text{if} & \ (\text{Abs} (\text{hitpos}.x() - \text{c1}.x()) < 1.\text{e}-6) \\
\quad \text{normal} & \ = \ \text{Vector}(-1,0,0); \\
\text{else if} & \ (\text{Abs} (\text{hitpos}.x() - \text{c2}.x()) < 1.\text{e}-6) \\
\quad \text{normal} & \ = \ \text{Vector}(1,0,0); \\
\text{else if} & \ (\text{Abs} (\text{hitpos}.y() - \text{c1}.y()) < 1.\text{e}-6) \\
\quad \text{normal} & \ = \ \text{Vector}(0,-1,0); \\
\text{else if} & \ (\text{Abs} (\text{hitpos}.y() - \text{c2}.y()) < 1.\text{e}-6) \\
\quad \text{normal} & \ = \ \text{Vector}(0,1,0); \\
\text{else if} & \ (\text{Abs} (\text{hitpos}.z() - \text{c1}.z()) < 1.\text{e}-6) \\
\quad \text{normal} & \ = \ \text{Vector}(0,0,-1); \\
\text{else} \\
\quad \text{normal} & \ = \ \text{Vector}(0,0,1)
\end{align*}
\]