HWRT

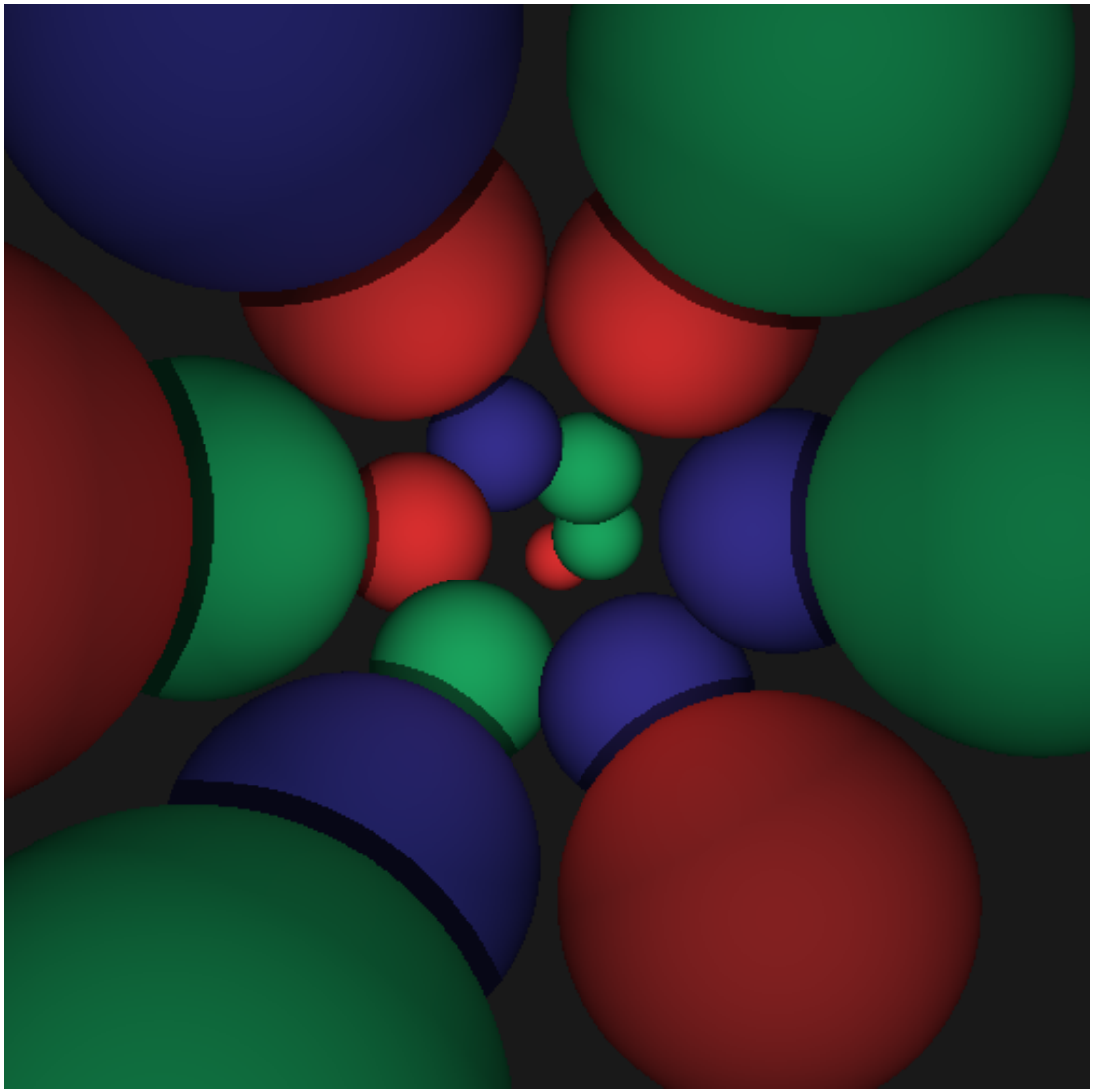
THE UNIVERSITY OF UTAH

CS 6958
LECTURE 10
LOADING THE SCENE

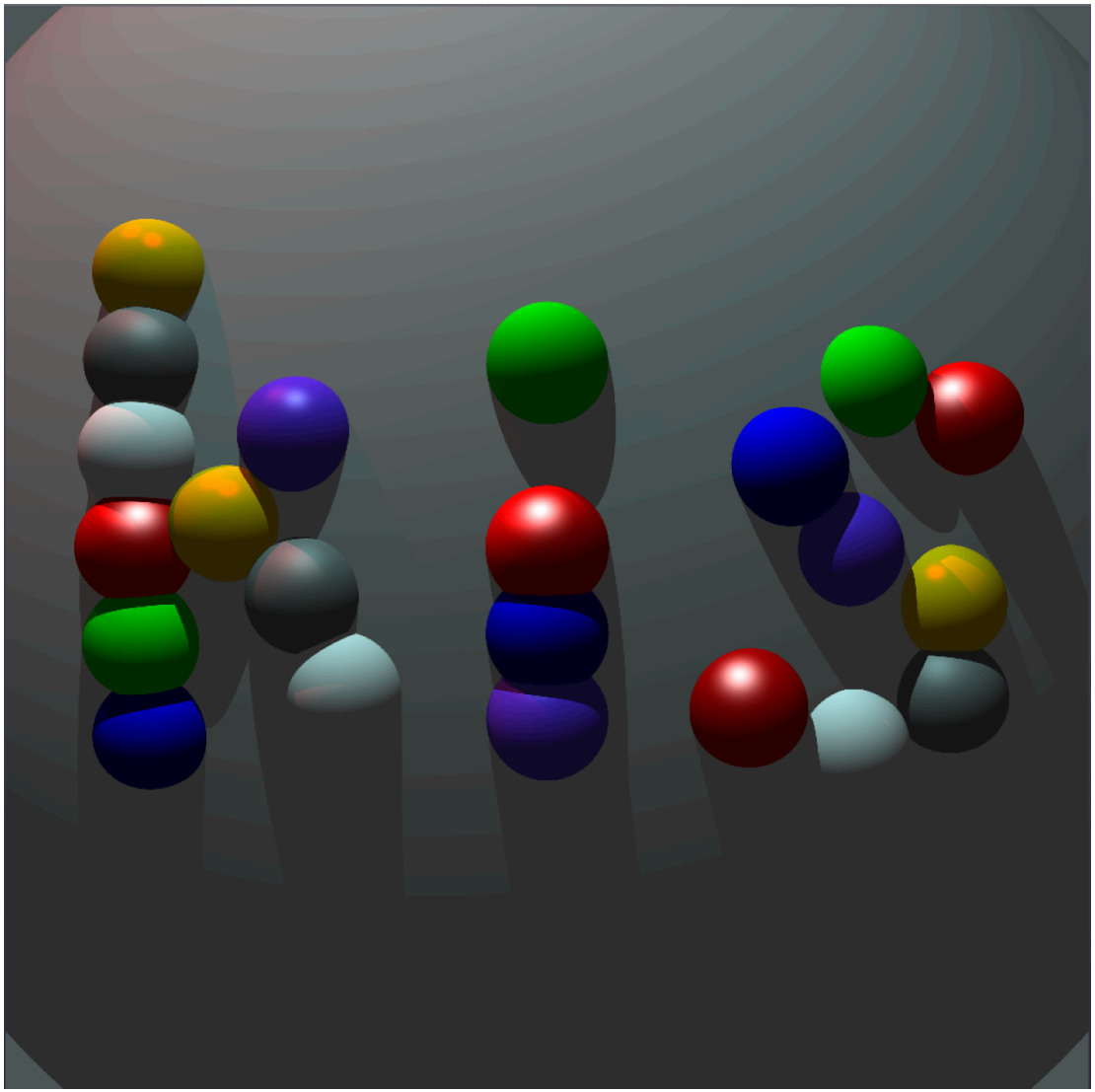
February 10, 2014

Creative!

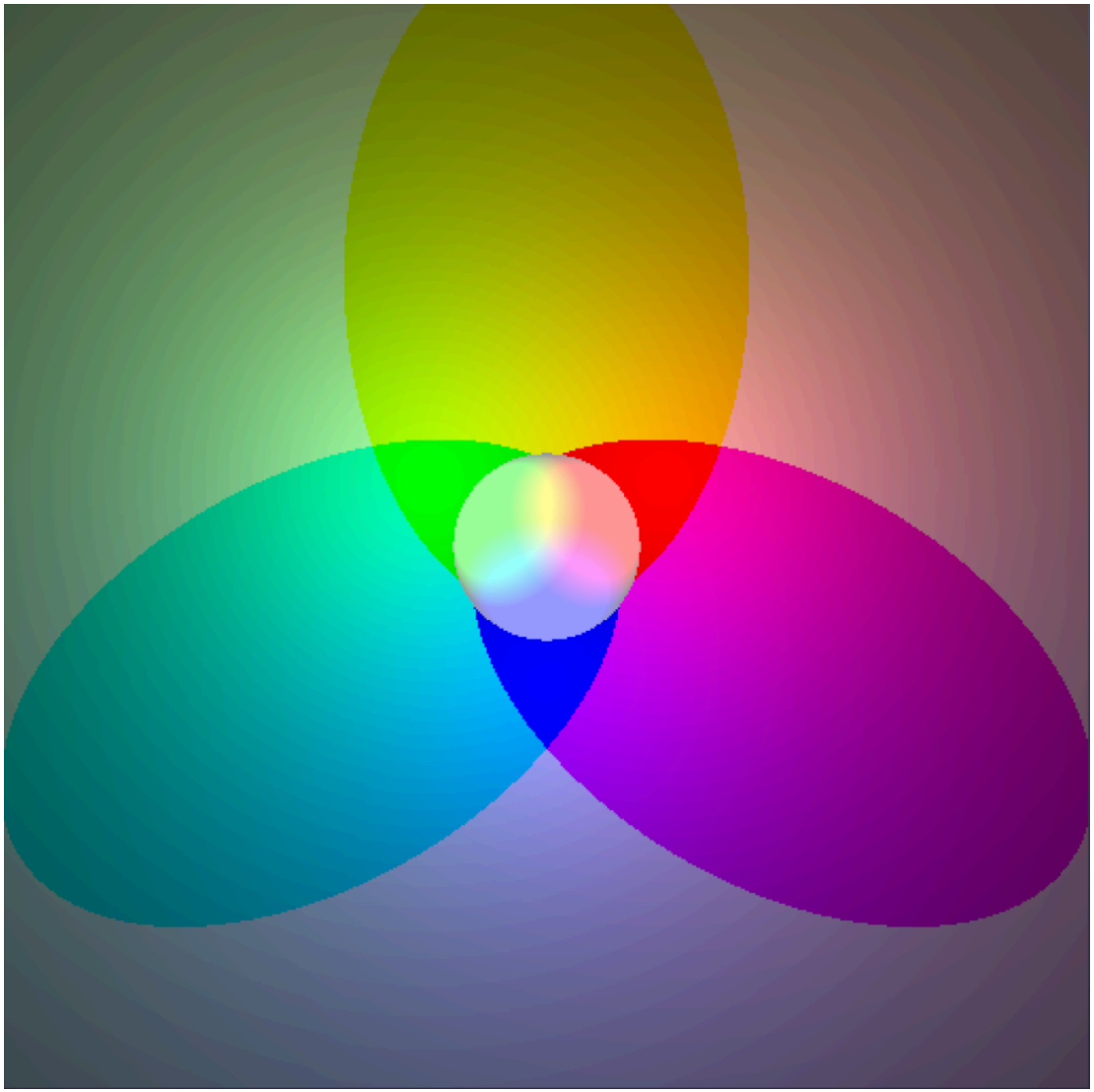
Creative!



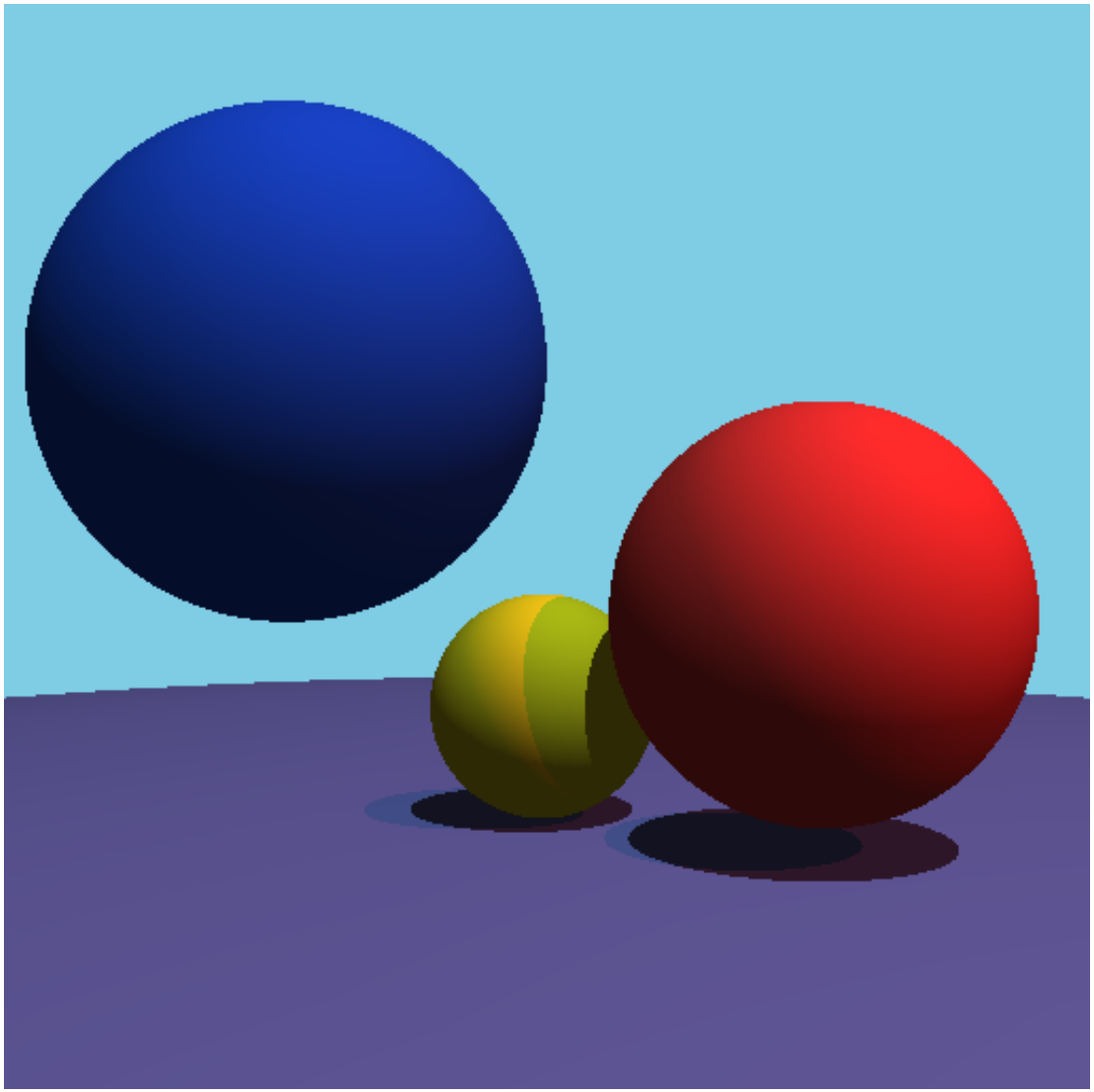
Creative!



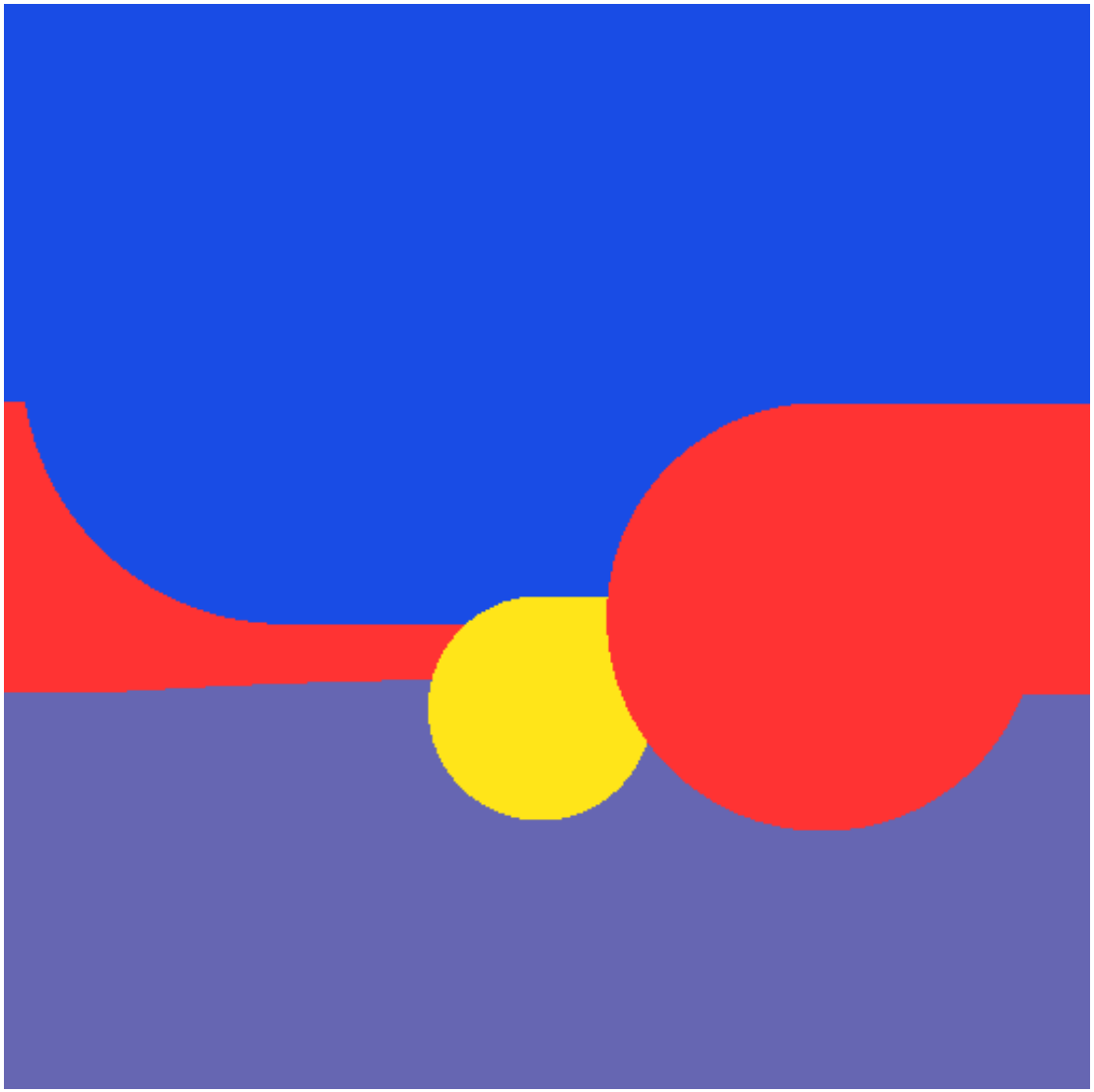
Creative!



Bloopers



Bloopers



Profiling / Reordering

- Examining one top offender instruction yields:
  - Reducing stalls by up to 3 cycles
  - Increasing performance by up to 0.5%
- Automated reordering (peephole optimizer)
  - Maybe 5% performance boost
- Real offender will be when LOADs take > 100 cycles

Branches in Profile

Assembly:	Num Executions	Data Stall Cycles	Stalls / Execution	
3126: FPDIV r22, r22, r8	28805	429893	14.92	?
3134: FPDIV r30, r26, r12	28805	403270	14	
3144: FPDIV r25, r25, r26	28805	383632	13.32	?
3123: FPINVSQRT r8, r26	28805	374465	13	

Profiled Block

```
        bneid     r3,     $0B_54
        FPDIV     r22,    r22,    r8
        brid      $0B_55
        FPRSUB    r3,     r22,    r30
$0B_54:
        FPADD     r3,     r30,    r22
$0B_55:
        ORI       r8,     r0,     .5
        FPMUL     r26,    r3,     r8
```

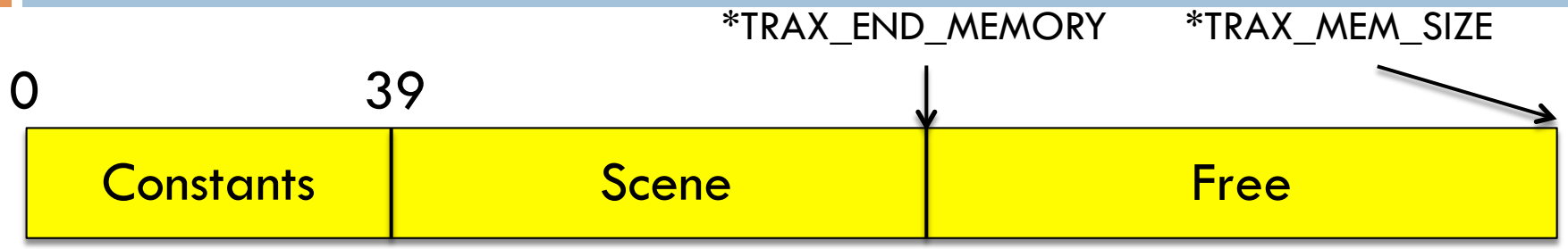
Execution Order 1

```
1          bneid     r3,     $0B_54
2          FPDIV     r22,    r22,    r8
           brid      $0B_55
           FPRSUB    r3,     r22,    r30
     $0B_54:
3          FPADD     r3,     r30,    r22
     $0B_55:
4          ORI       r8,     r0,     .5
5          FPMUL     r26,    r3,     r8
```

Execution Order 2

```
1           bneid     r3,     $0B_54
2           FPDIV     r22,    r22,    r8
3           brid      $0B_55
4           FPRSUB    r3,     r22,    r30
    $0B_54:
            FPADD     r3,     r30,    r22
    $0B_55:
5           ORI       r8,     r0,     .5
            FPMUL     r26,    r3,     r8
```

Recap: Global Memory



- loadi, loadf, storei, storef
- Lots of data pre-loaded
- --view-file
  - Camera data
- --model
  - Geometry (.obj or .iw format)
  - BVH info (built from geometry)
  - Material info (.obj files specify a .mtl file)
- --light-file
  - Light

Recap: Data Pointers

- Light — loadi(TRAX_START_LIGHT)
- Camera — loadi(TRAX_START_CAMERA)
- Model
  - BVH/Grid — loadi(TRAX_START_SCENE)
  - Triangles — loadi(TRAX_START_TRIANGLES)
  - Vertex normals — …
  - Texture coordinates — …
- Materials — loadi(TRAX_START_MATLS)

Light Files (--light-file)

- Very simply format (too simple)
- 3 floats specifying the location of a point light

278.0 279.5 540.0

Loading Lights From Memory

- Old constructor:

```
PointLight(const Vector& position, …)
```

- New constructor:

```
PointLight(int addr)
{
    position = Vector(loadf(addr, 0),
                      loadf(addr, 1),
                      loadf(addr, 2));
    ...
}

PointLight theLight(loadi(TRAX_START_LIGHT));
```

Helpers

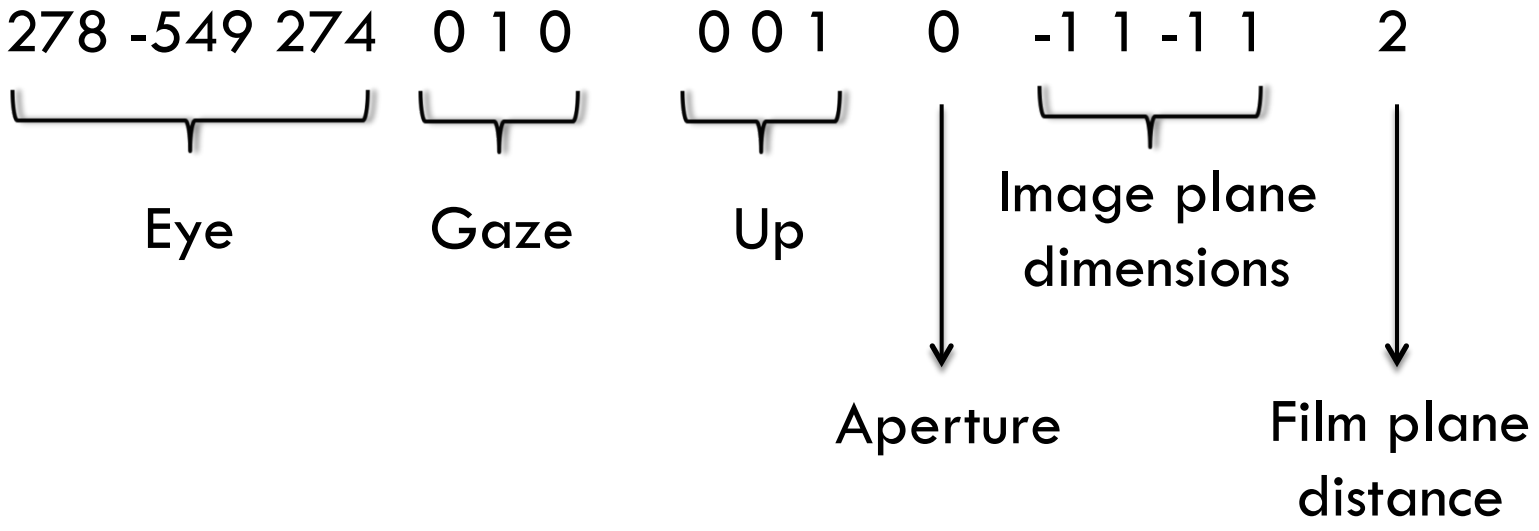
- In Vector.h

```
inline Vector loadVectorFromMemory(
const int &address)
{
  return Vector(loadf(address, 0),
                loadf(address, 1),
                loadf(address, 2));
}
```

- Similarly for Color, …

View Files (--view-file)

- Simple text format (details soon)



- Simulator builds and loads camera using this info

Loading the Camera

- start_camera = loadi(TRAX_START_CAMERA)

- eye[x, y, z] is at start_camera + [0..2]
- (deprecated): start_camera + [3..8]
- up: start_camera + [9..11]
- gaze: start_camera + [12..14]
- u: start_camera + [15..17]
- v: start_camera + [18..20]
- Advanced info: start_camera + [21..33]

Loading the Camera

- Recommend a constructor that takes an address

```
PinholeCamera::PinholeCamera(int addr){
  eye      = loadVectorFromMemory(addr + 0);
  up       = loadVectorFromMemory(addr + 9);
  gaze     = loadVectorFromMemory(addr + 12);
  u        = loadVectorFromMemory(addr + 15);
  v        = loadVectorFromMemory(addr + 18);
}

PinholeCamera camera(loadi(TRAX_START_CAMERA));
```

Triangles

- 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)
- Deprecated (address + 9)
- material ID (address + 10)

Triangle Constructor

- Recommend:

```
Tri(int address)
{
    load 3 vertices
    load mtl ID
    ID = address;
}
```

Don't call your class "Triangle"!

- Save the address as the ID
  - Save this ID in the HitRecord
  - In case we need it later (depends on implementation)

Triangle Normals

```
inline Vector normal() const
{
  Vector edge1 = p1 - p3;
  Vector edge2 = p2 - p3;
  Vector n = Cross(edge1, edge2);
  n.normalize();
  return n;
}
```

- Don't compute normal unless you need to shade this triangle
- Consider saving e1 and e2 during intersection
  - Then pass HitRecord to normal()

Avoid Memory Traffic

- Should we load the material ID when checking for intersection?

- In BVH traversal:

```
if leaf node
    for all primitives in leaf
        Tri t(address)
```

Load in constructor? (save in HitRecord)
Likely to be cached anyway

- In shade():

```
mtl_ID = loadi(hitrecord.getObjID(), 10)
```

Or load only when needed?

mtl_id at tri address + 10

Avoid Memory Traffic

- Save as much as possible in HitRecord
- Recommend:
  - e1, e2 (computed during intersection) → use to compute normal
  - Material ID
- Alignment
  - 11 words is very odd size for 16-word cache lines
  - Memory loader doesn't do any padding for triangles

Better(?) Triangle Normals

```
inline Vector normal(const HitRecord& hr)
  const
{
  Vector n = Cross(hr.edge1, hr.edge2);
  n.normalize();
  return n;
}
```

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

```
int start_tris = GetStartTriangles();
int num_tris = GetNumTriangles();
for(int i=0; i < num_tris; i++)
 {
   Tri t(start_tris + (i * 11));
   t.intersect(hitRec, ray);
 }
```

- Recall that Tri constructor takes address, generates loads

Materials

- Materials are 25 words each, starting at:
  - `int start_matls = GetMaterials();`

```
int shader_id = hitRec.getMtlID();
int shader_addr = start_matls + (shader_id * 25)
```

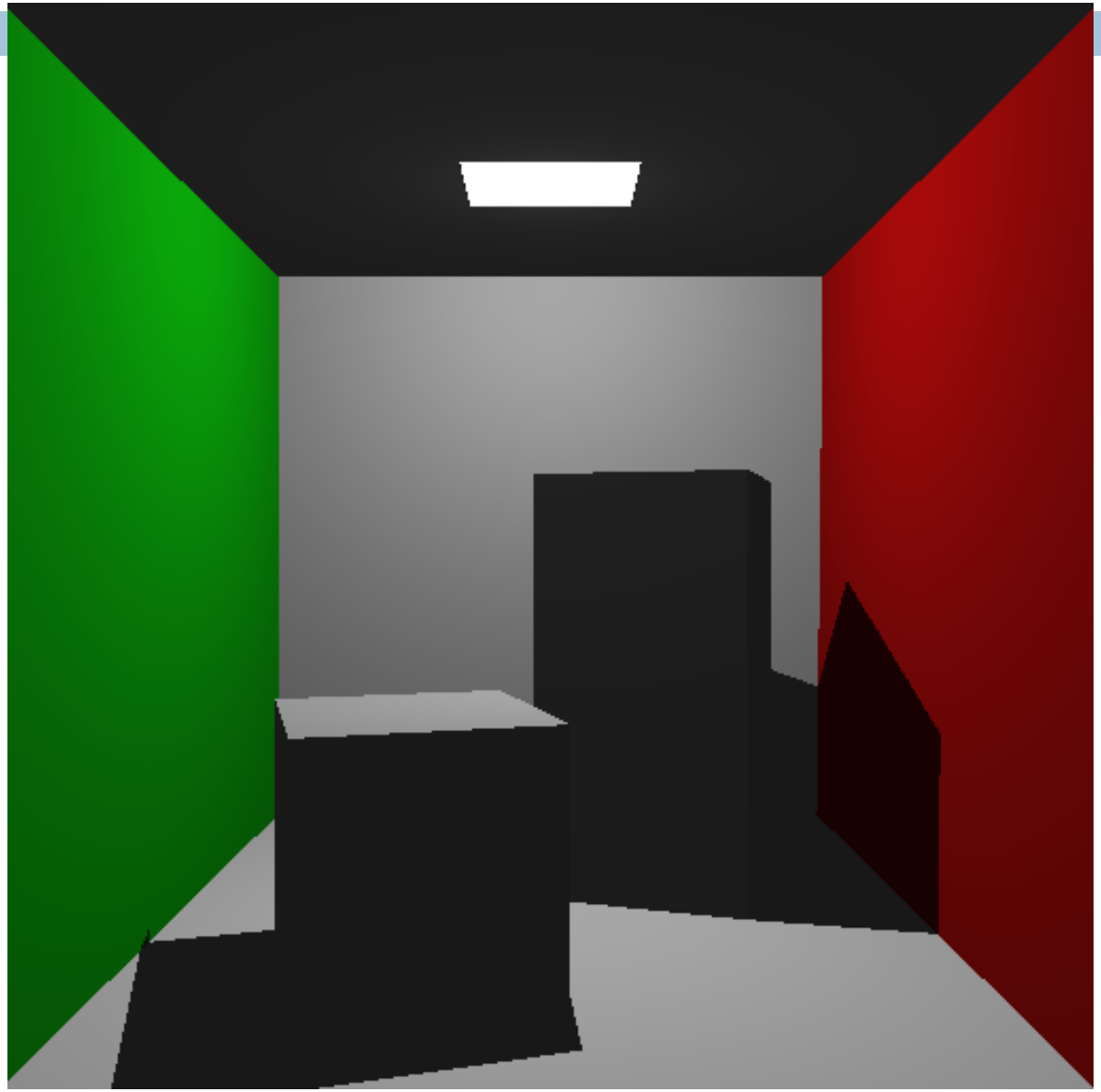
Materials

- The diffuse color of each material is what we care about for now

- Diffuse color is stored in the material at offsets:
  - 4(r), 5(g), 6(b)

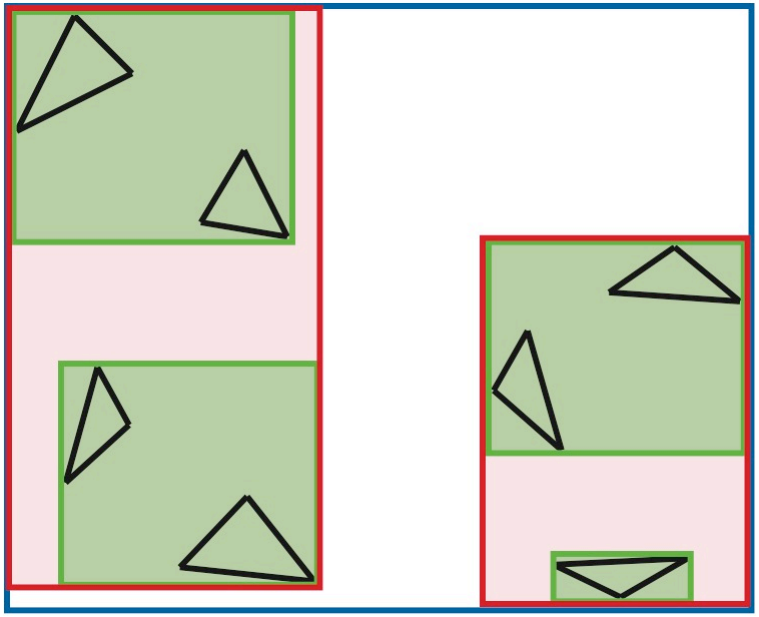
-
```
Color diffuse(loadf(shader_addr, 4),
              loadf(shader_addr, 5),
              loadf(shader_addr, 6));
```

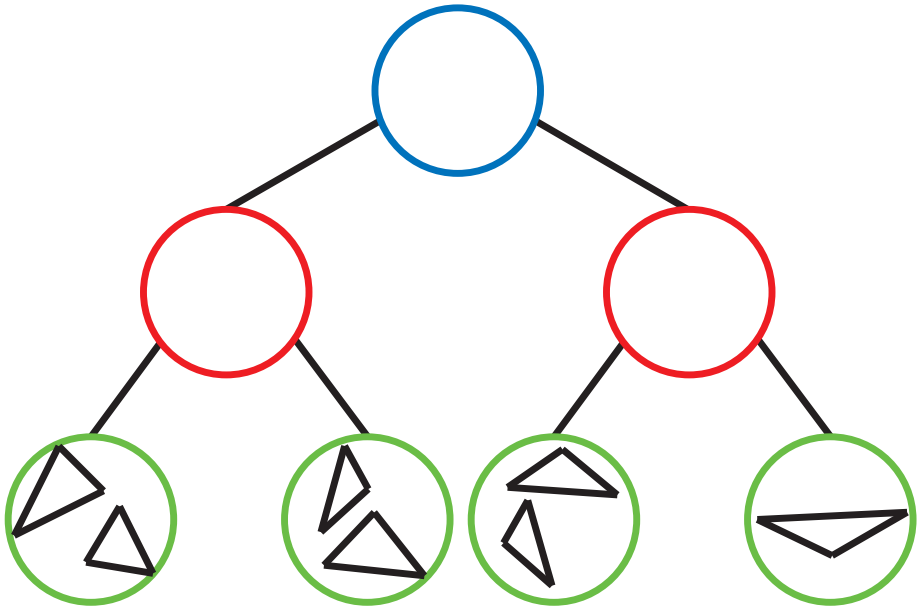
Assignment 3 (part 1)



Recap: BVH

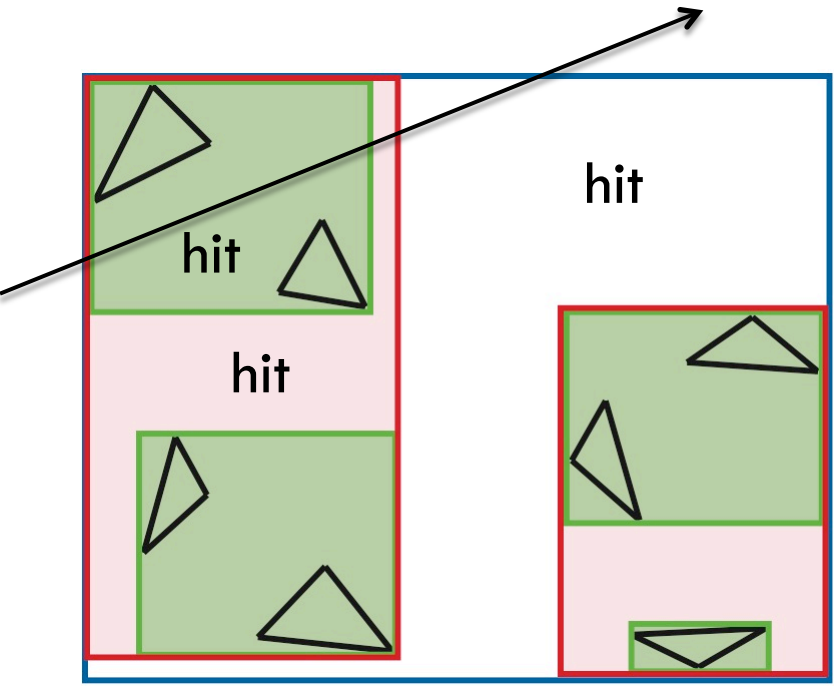
- Facilitates fast ray-geometry intersection
  - Try to avoid obviously missed geometry

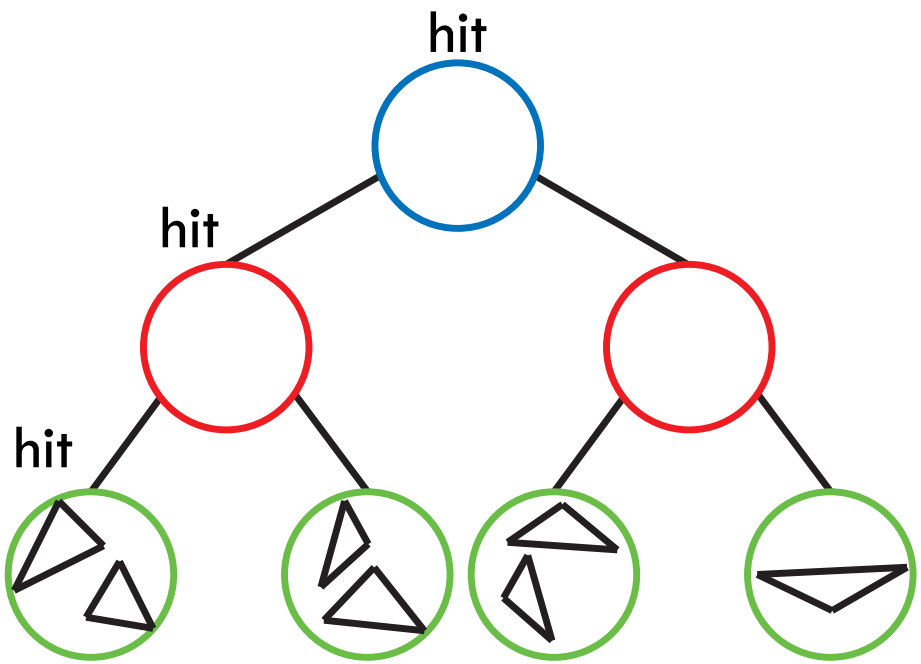




Recap: BVH

- Facilitates fast ray-geometry intersection
  - Only traverse branches that the ray hits
  - Basically depth-first traversal (conditional)





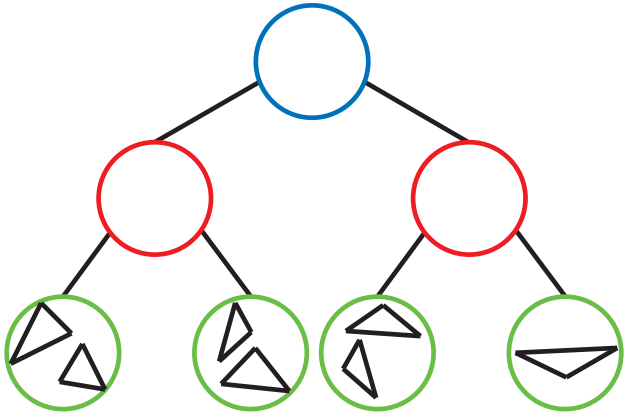
Traversing the BVH

- No recursion!
  - Stack frames will quickly outgrow local memory space
  - Function calls are expensive
- But tree traversal is inherently recursive
- Use a software-managed stack

```
int stack[32]; // holds node IDs
int sp = 0;    // stack pointer
```

Pseudo code

```
current = root ID (0)
while(true)
    if(ray intersects current node)
       if(interior node)
             push rightChildID
             current = leftChildID
             continue;
       else
             intersect-test all triangles in leaf
    if(stack is empty)
       break;
    current = pop stack
```



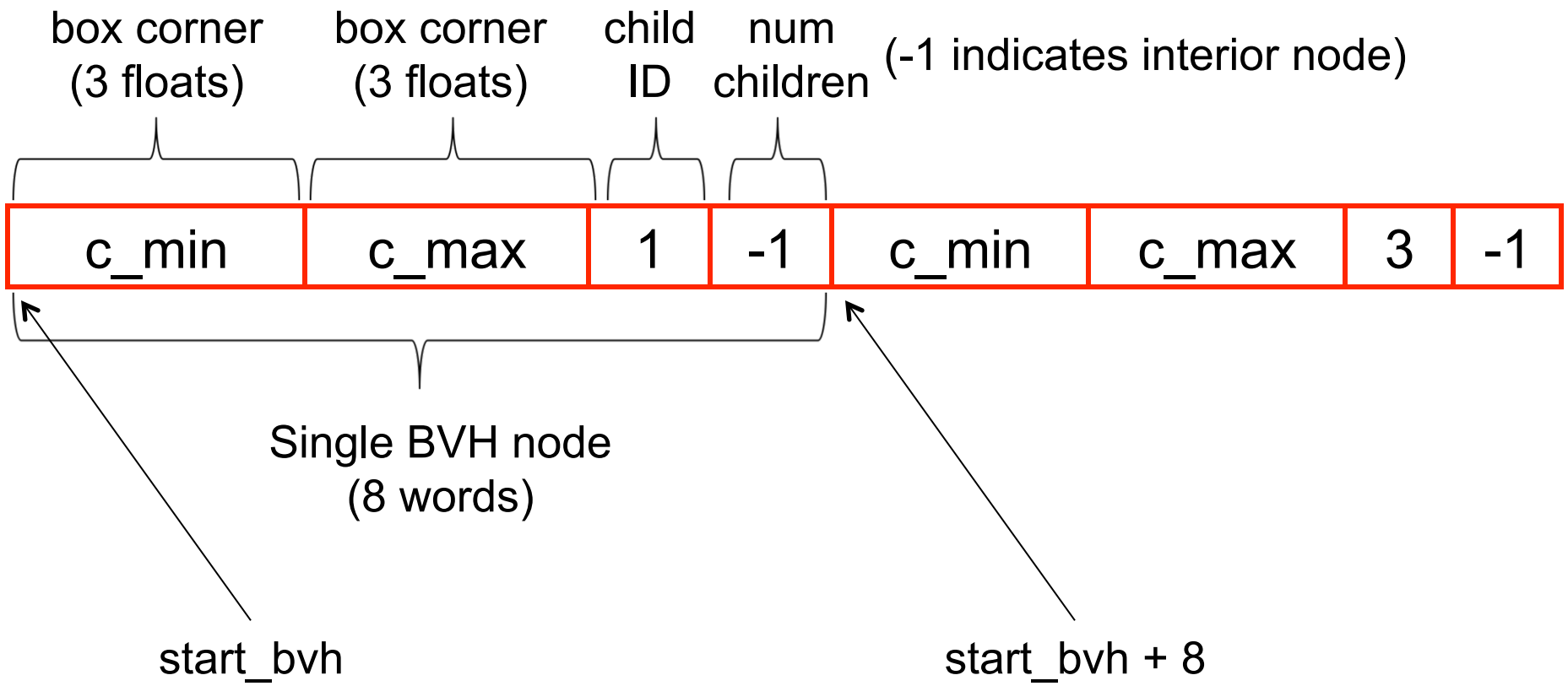
Box Intersection

```
Box::intersect(HitRecord& hit, const Ray& ray) const{
    float tnear, t2;
    Vector inv = 1.f / ray.direction();
    Vector p1 = ( c1 - ray.origin() ) * inv;
    Vector p2 = ( c2 - ray.origin() ) * inv;
    Vector mins = p1.vecMin( p2 );
    Vector maxs = p1.vecMax( p2 );
    tnear = max( mins.x(), max( mins.y(), mins.z()));
    t2 = min( maxs.x(), min( maxs.y(), maxs.z()));

    if(tnear < t2) ...
```

Make sure to account for inside hits!

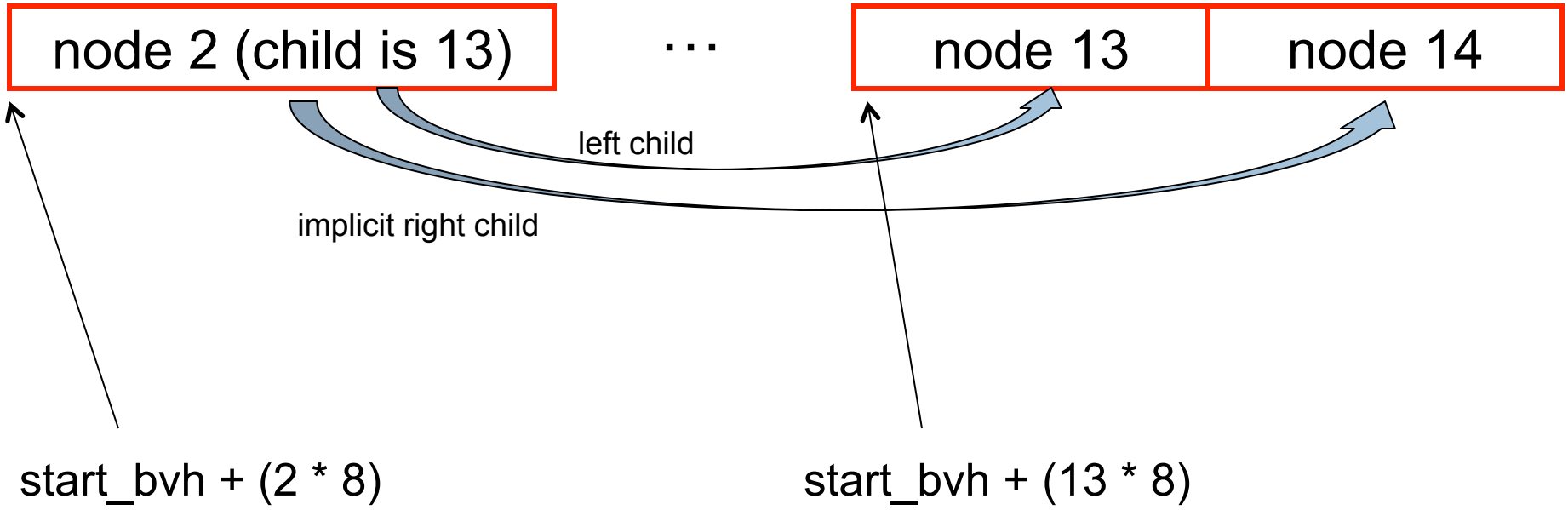
BVH layout

- `int start_bvh = GetBVH();`

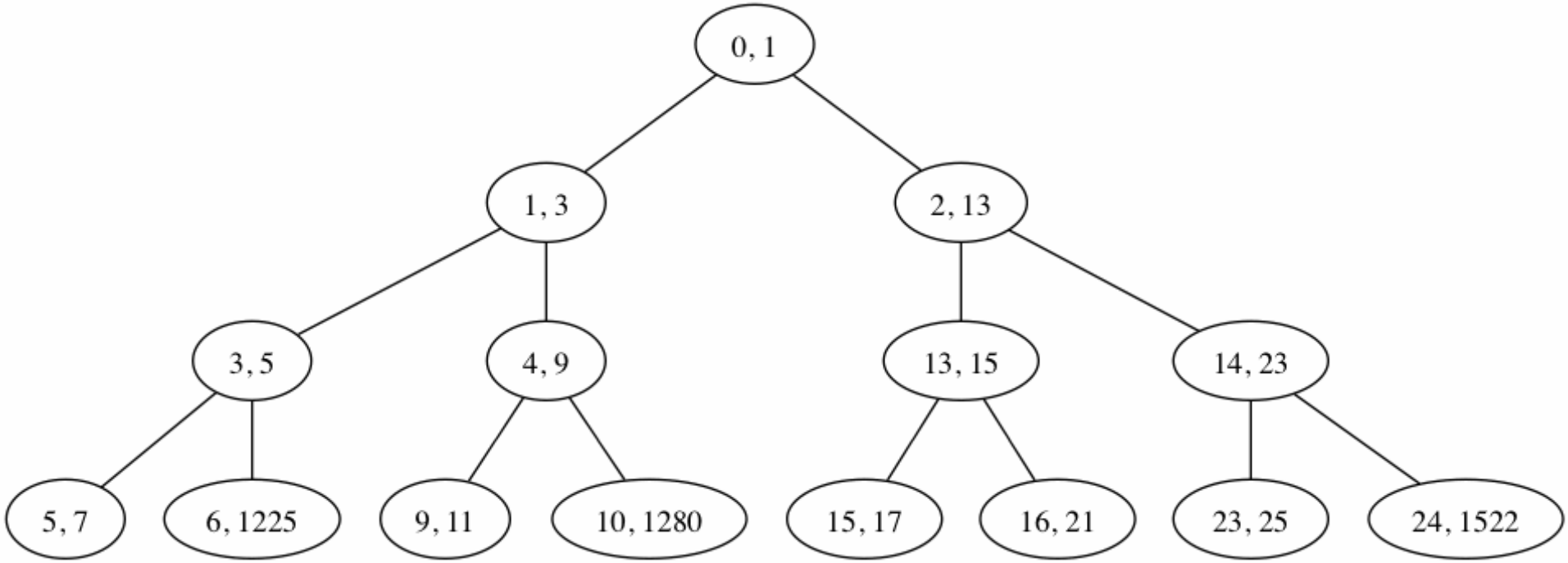


BVH layout

- Sibling nodes are next to each other in memory
- Right child's ID is always left_id + 1



BVH layout


0, 1
1, 3
2, 13
3, 5
4, 9
13, 15
14, 23
5, 7
6, 1225
9, 11
10, 1280
15, 17
16, 21
23, 25
24, 1522

BVH Nodes

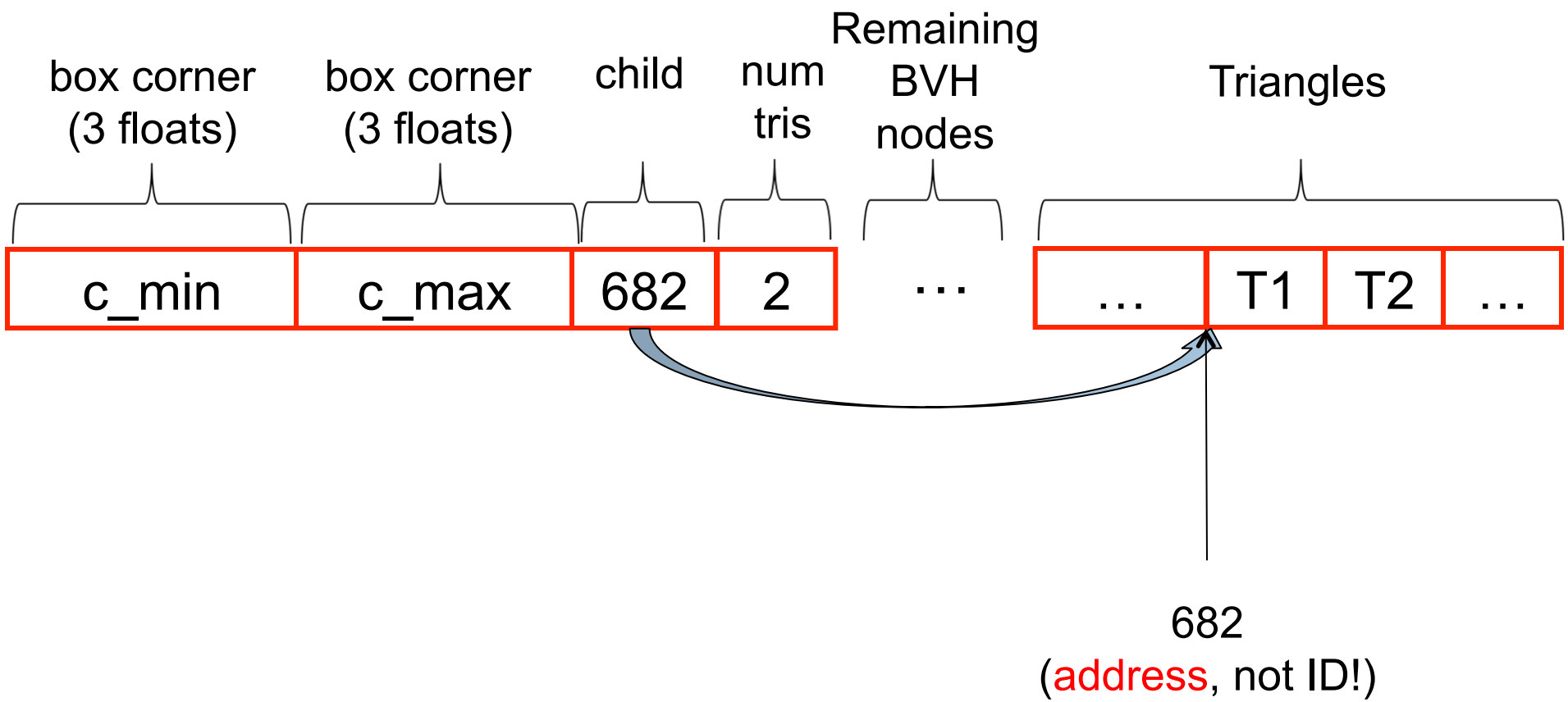
- As with all data held in global memory, recommended:

```
BVHNode::BVHNode(int addr){
  box.c1 = loadVectorFromMemory(addr + 0);
  box.c2 = loadVectorFromMemory(addr + 3);
  num_children = loadi(addr + 6);
  child = loadi(addr + 7);
}
```

Leaf Nodes

- Implied differently:
  - num_children > 0
  - child_ID = *address* of node's first triangle
    - Not ID of first triangle!
    - Leaf node's triangles are consecutive in memory

Leaf Nodes


box corner
(3 floats)
box corner
(3 floats)
child
num
tris
Remaining
BVH
nodes
Triangles
c_min
c_max
682
2
…
…
T1
T2
…
682
(address, not ID!)

Example

```
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;
      BVHNode node(node_addr);
      HitRecord boxHit;
      node.box.intersect(boxHit, ray);
      if(boxHit.didHit())
     // and so on...
```

Example (continued)

```
left_id = node.child;
if ( node.num_children < 0 ) // interior node
     {
       stack[ sp++ ] = left_id + 1;
       continue;
     }
// leaf node
tri_addr = left_id;
for ( int i = 0; i < node.num_children; ++i)
   {// intersect triangles}
// ... finish outer loop, manage stack
```

BVH Implementation

- My bvh class contains just a pointer to start_bvh

```
BoundingVolumeHierarchy(const int &_start_bvh)
    {
      start_bvh = _start_bvh;
    }
```

- Nodes are loaded one at a time as needed

BVH Implementation

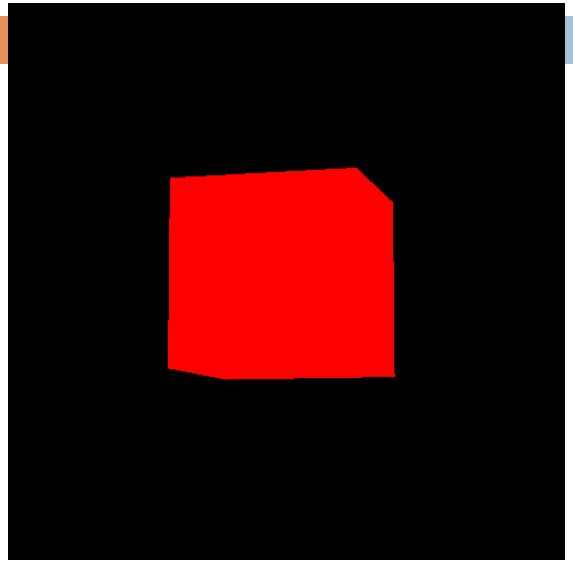
```
inline void intersect(HitRecord& hit,
                      const Ray& ray) const
```

- 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!

Big Picture

```
for each pixel...
 Ray ray;

 camera.makeRay(ray, x, y);

 HitRecord hit;

 scene.bvh.intersect(hit, ray);

 result = shade(...);
```

Program 3 (part 2)



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
- Can be tricky!
  - What if node intersection occurs inside box?