HitRecord

- Need some way of keeping track of closest hit
- Recommend a data structure to hold:
  - closest intersection (float distance)
  - Object ID of closest hit object
- HitRecord::HitRecord(const float max)
  - will not consider any hits farther away than max

HitRecord

bool hit(float t, int obj_id)

Use like:

hitrecord.hit(t, i)

Bool return value for optimizations (and more uses later)

HitRecord

- float minT()
  - returns minimum T value
- bool didHit()
  - returns whether or not anything was hit
- int objID()
  - returns ID of closest hit object

- Use a single hit record for each pixel (or for each shading point)

Improved Sphere

- inline void intersect(HitRecord& hit, const Ray& ray) const
- Need to pass a HitRecord to any intersection function
- No longer need to return true/false

Improved Sphere

- Add method:
  - inline Vector normal(const Vector &hitPoint) const;
  - returns the normal, given a hit position
- use HitRecord and original ray to determine hit point
  - hitPoint = ray.origin + ray.direction * hr.minT()

Improved Sphere

- Needs to keep track of its “material”
- In TRaX, all materials are loaded for you
- We just keep track of an int: matl_id
- Sphere(const Vector& center, const float radius, const int id, const int mat_id);

Data structures: Light

- Method to compute light incident to a point
  - Returns Color and Direction
- Point light:
  - Position
  - Color
- Directional light:
  - Vector direction
  - Color

PointLight

```
float getLight(
        Color& light_color,
        Vector& light_direction,
        const Vector& hitPoint) const;
```

- return value is distance to light
  - (pass this as max to HitRecrod constructor)
- light_direction is normalized

Data structures: Camera

- Method to compute ray for image x,y [-1,1]
- Pinhole camera:
  - Position
  - Lookat point or gaze direction
  - Up vector
  - U_length
  - Aspect ratio

PinholeCamera

```
PinholeCamera(const Vector& eye, const Vector&
    lookat, const Vector& up, const float ulen, const float
    aspect_ratio);

void makeRay(Ray& ray, float x, float y) const;
```

Program 2

Part 1:

Run your program 1 on the TRaX simulator

./simhwrt --no-scene --load-assembly ../prog01/rt-llvm.s

generates “out.png” (and prints a whole bunch of stats)

Rembmer to follow TRaX programming guidelines!

Running on TRaX

- First get the CPU version working (if it doesn't work, trax will definitely not work)
- If the trax version doesn't work, there are 2 likely culprits:
  - The assembler failed
  - The broken calling convention failed

Running on TRaX

If the assembler failed, your code either uses the double data type, or tries to call some stdlib function, such as “new” or “malloc”

ERROR: undefined symbol: __malloc

failed to add instruction: bneid

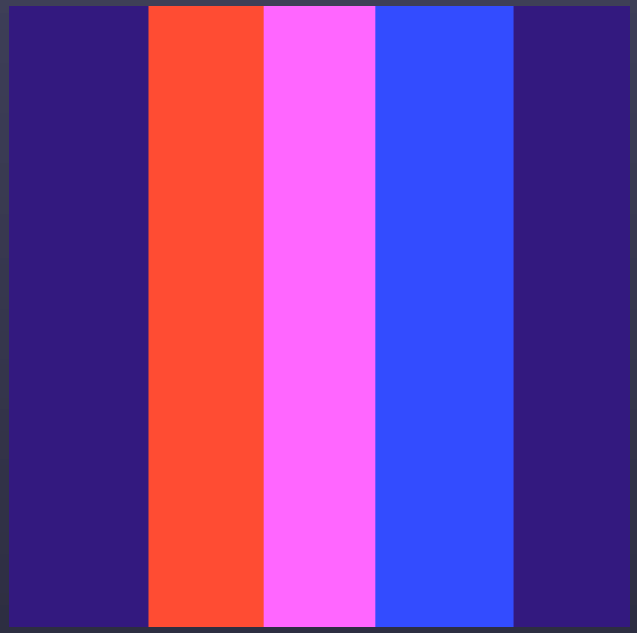
line 34

assembler returned an error, exiting

Running on TRaX

If the simulator runs, but produces incorrect output, you most likely are not passing some variable by reference:

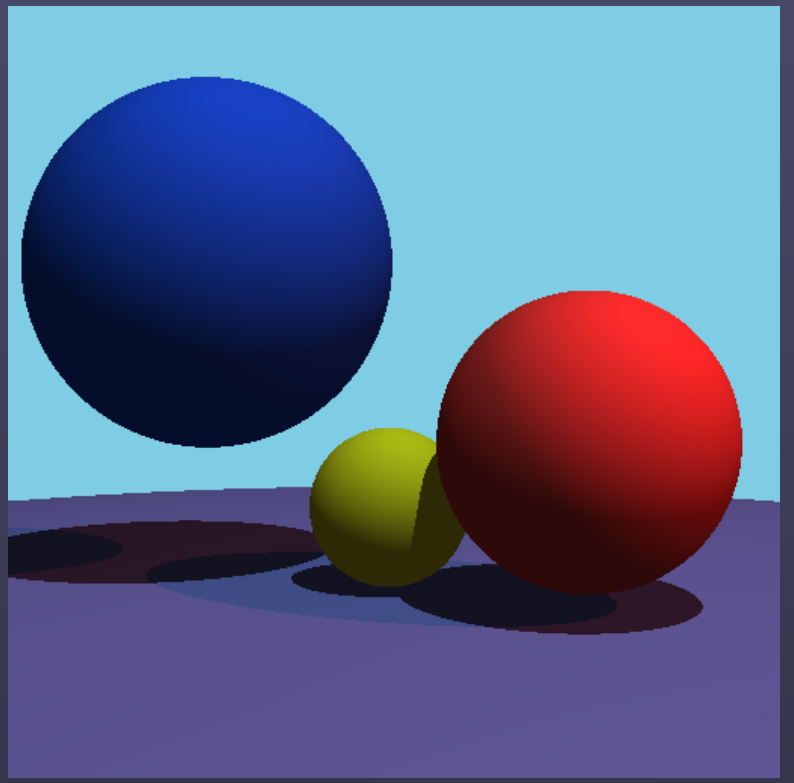
Ray::Ray(Vector p0, Vector d) ← not reference!



Program 2

- Part 2: Complete ray tracer with shadows
- Creative: better test scene

- Will be posted as soon as possible (no later than Thurs)



Program 2

- For now, just keep an array of spheres, colors, and lights, pass them to shading functions

```
Color materials[4];
Sphere spheres[4];
PointLight lights[2];
materials[0] = Color(0.1f, 0.3f, 0.9f);
…
spheres[0] = Sphere(Vector(1.5f, 3.5f, 4.f), 2.4f, 0, 0);
…
result = shade(hitRecord, ray, hitPoint, spheres, materials,
    lights);
```

Static scenes

- Alternatively, keep a “Scene” data structure, with pointers to these arrays (keep method signatures clean)
- The memory for these must never go out of scope (statically allocate them in main)
- Later on, all memory will be handled by the simulator

Watch out for

- The TRaX simulator has several classes that may conflict with your class names:
- Camera
- Material
- Primitive
- Triangle
- Vector3

- Don’t use these as class names in your ray tracer code