Creative
Creative
Creative
Creative
Creative
Accidental Art
Accidental Art
Lab 1 – Perf/Area Scaling
Lab 1 - Performance

Single-thread Performance

<table>
<thead>
<tr>
<th>Person</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120</td>
</tr>
<tr>
<td>2</td>
<td>130</td>
</tr>
<tr>
<td>3</td>
<td>360</td>
</tr>
<tr>
<td>4</td>
<td>370</td>
</tr>
<tr>
<td>5</td>
<td>380</td>
</tr>
<tr>
<td>6</td>
<td>390</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
</tr>
<tr>
<td>8</td>
<td>410</td>
</tr>
<tr>
<td>9</td>
<td>420</td>
</tr>
</tbody>
</table>
Lab 1 - Performance

Total Instructions

<table>
<thead>
<tr>
<th>Person</th>
<th># instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50000000</td>
</tr>
<tr>
<td>2</td>
<td>50000000</td>
</tr>
<tr>
<td>3</td>
<td>10000000</td>
</tr>
<tr>
<td>4</td>
<td>10000000</td>
</tr>
<tr>
<td>5</td>
<td>10000000</td>
</tr>
<tr>
<td>6</td>
<td>10000000</td>
</tr>
<tr>
<td>7</td>
<td>10000000</td>
</tr>
<tr>
<td>8</td>
<td>10000000</td>
</tr>
<tr>
<td>9</td>
<td>10000000</td>
</tr>
</tbody>
</table>
Lab 1 - Performance

- Avg increase in FPS (1 → 8 threads): 7.6x
- Best FPS/area config (6122 FPS / sq mm):
  - 1 icache, 4 banks
  - FPADD 2 3
  - INTADD 1 2
  - BLT 1 2
  - BITWISE 1 3
  - 1 of everything else
  - Other configs were all similar
- Will get much more interesting when memory is involved
Resource Conflicts

- If 2 threads conflict, one will naturally become out of sync with the other
- Will no longer try to issue to same bank on next cycle
- `icache bank = PC % num_banks`

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Thread 1 PC</th>
<th>Thread 2 PC</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>32</td>
<td>Conflict Thread 2 stalls</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>32</td>
<td>Both issue</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>33</td>
<td>Both issue</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td>34</td>
<td>Both issue</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>35</td>
<td>Both issue</td>
</tr>
</tbody>
</table>
Why 4 icache banks?

- Each bank is double pumped (2 services/cycle)
  - 4 banks can service at most 8 threads / cycle

- 8 threads total
  - Naturally out of sync

- On average, 1 instruction returned per thread
**SPMD Execution**

Lack of synchronization can be a good thing!
Transforms

- The two performance outliers:
  - Spheres are defined with a transformation matrix
  - In Sphere::intersects
    - `Ray r = xform.ToNodeCoords(ray);`

- Others are all defined in world space
  - However, eliminates possibility of “instancing”
Transforms

profile(0);
Ray r = xform.ToNodeCoords(ray);
profile(0);

- This line of code accounts for 81% of all cycles
- Transform is roughly 3x more expensive than ray-sphere intersection
Data Storage (Vector, Color)

- Erik’s
  - float x, y, z

- Mine
  - float data[3]
Compiler likes to put arrays on stack.

Add, addi come from array offset calculation.
Recap: Direct + “Ambient” Light
Better Lighting – Coming Soon
Lambertian Shading

Let:
- $V =$ ray direction
- $O =$ ray origin
- $t =$ distance to hit object
- $N =$ surface normal of hit object
- $L =$ vector from hit point to light

if camera ray did not hit anything
return background color
else
see next slide…
Lambertian Shading

\[ P = O + tV \]  \hspace{1cm} // \text{hit point}

call primitive to get normal \( N \)

\[ \text{costheta} = N \cdot V \]

if (\text{costheta} > 0.\text{f}) \hspace{1cm} // \text{is normal flipped?}

\[ \text{normal} = -\text{normal} \]

Color \( \text{light} = \text{ambient} \ast \text{Ka} \)  \hspace{1cm} // \text{start with ambient light}

foreach \( \text{light} \)

get \( \text{lightColor} \) and \( L \)

\[ \text{dist} = L.\text{length}(); \quad L.\text{normalize}(); \]

\[ \text{cosphi} = N \cdot L \]

if(\text{cosphi} > 0.\text{f}) \hspace{1cm} // \text{is light on right side of object?}

if no intersection with \( 0 < t < \text{dist} \) \hspace{1cm} // \text{do we have sight of the light?}

\[ \text{light} += \text{lightColor} \ast (\text{Kd} \ast \text{cosphi}) \]  \hspace{1cm} // \text{add light’s color}

\[ \text{result} = \text{light} \ast \text{surface color} \]  \hspace{1cm} // \text{multiply all light by object color}
Types of Lights

- There are plenty of light models
  - We will mostly use point lights
  - Others: area lights, emissive materials
Light Implementation

class Light{
    char type; ← Optional: (point, directional, etc)
    Vector position;
    Color color;
    ...
    getLight(const Vector& hitpos, ...) const;
}

- Multiple ways to implement getLight
- We need its color, a vector pointing from the hit-point to the light, and the distance
getLight Recommendation

float getLight(const Vector& hitpos,
               Color& light_color,
               Vector& light_direction) const

- Returns distance
- Sets color via reference
- Sets normalized direction via reference
Sphere Normals

- How can we find the normal of a sphere at a point P on its surface?
- Normal has same direction as (P-C)
  - Just normalize and return it
- inline Vector Sphere::normal(const Vector &hitPoint) const
Hit Record

- We need some way of keeping track of closest hit

- Recommendation: some data structure to hold:
  - Closest hit distance
  - Closest object ID
  - Others?
    - Normal information
    - Barycentric coordinates
    - ...

Hit Record

- HitRecord::HitRecord(const float max_t)
  - Ignores intersections further away than max_t
  - Useful for shadow rays
  - Use “infinity” for other rays

- bool HitRecord::hit(distance, objectID)
  - Use inside intersection routine, i.e.:
  - if(discriminant > 0.f) hr.hit(…)

*Hit Record*
Hit Records

- float getMinT()
- bool didHit()
- int getObjID()

Use a single HitRecord for each ray
  - You don’t want shadow rays overriding camera rays!
Improved Spheres

- Need a little more information now

Sphere::Sphere(
    const Vector& center,
    const float radius,
    const int obj_id,
    const int mat_id)

inline void Sphere::intersect(
    HitRecord& hit,
    const Ray& ray) const
Cameras – Map Pixels to Rays

Create scene
Preprocess scene
foreach pixel
foreach sample
  generate ray
Camera models

- **Typical:**
  - Orthographic
  - Pinhole (perspective)

- **Advanced:**
  - Depth of field (thin lens approximation)
  - Sophisticated lenses (“A realistic camera model for computer graphics,” Kolh, Mitchell, Hanrahan)
  - Fish-eye lens
  - Arbitrary distortions
  - Non-visible spectra
    - Wi-Fi/radio antenna
Camera Models

- Map pixel coordinates to [-1 to 1]
- Pay careful attention to pixel centers
- Feed x, y to camera to generate ray

- Non-square images
  - Camera knows about aspect ratio
  - Applies appropriate scaling
Orthographic projection

- “Film” is just a rectangle in space
- Rays are parallel (same direction)
Orthographic projection

- Defined as
  - a center $P$
  - two vectors $u$, $v$
- Ray origin $= P + xu + yv$
  - $x, y = [-1 .. 1]$ pixel coordinates
- Ray direction $= u \times v$
Pinhole camera

- Most common model for ray tracing
- Image is projected upside down onto image plane
- Infinite depth of field
Pinhole camera

- In software we invert this model
- Focal point is now called the eye point
Pinhole camera

- **E**: eye point
- **Up**: up vector (unit length)
  - Specifies orientation
- **θ**: field of view
- **Gaze**: looking direction
What we need

- What’s missing is $u, v$
  - These define the film plane
  - Not unit length

- Find them using
  - Gaze
  - Up
  - $\theta$
Finding $u, \ v$

$$\text{Gaze.normalize}()$$
$$u = \text{Gaze} \times \text{Up}$$
$$v = u \times \text{Gaze}$$

Use $\tan(\theta)$ to find length of $u$

OR:
Supply $u\_\text{len}$ as a parameter instead of $\theta$
Finding $u, v$

$Gaze.normalize()$

$u = Gaze \times Up$

$v = u \times Gaze$

$u.normalize()$

$v.normalize()$

$u *= u\_len$

$v *= (ulen / aspect\_ratio)$
Aspect ratio

- Aspect ratio = \( \frac{x_{\text{res}}}{y_{\text{res}}} \)

- Supply this to the camera as well
Camera Parameters

PinholeCamera::PinholeCamera(
const Vector& eye,
const Vector& gaze,
const Vector& up,
float u_len,
float aspect_ratio)

Derive and save u, v, normalized gaze
Generating a Ray

```cpp
void PinholeCamera::makeRay(
    Ray& ray,
    float x, float y) const

    origin = E
    direction = Gaze + xu + yv
    direction.normalize()

    x, y = [-1 .. 1] pixel coordinates
```
Field of View (defined by $u_{len}$)

28 deg  
60 deg  
108 deg
In TRaX

- Each thread will save a copy of the camera in local stack space
  - Cameras are about 12 words of data

- Don’t want to load camera from main memory every time we generate a ray