UU Graphics

academic year 2013/14 – 4th period

Theoretical Assignment #5:
Rasterization & Shading

June 10 2014

Assignment #1: *Rasterization*

You are given a triangle with projected vertices

$$\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3 \in \mathbb{R}^2.$$

These points are already given in screen coordinates [pixels]; however, the points can be placed anywhere – no guarantees. Further, your screen consists of $w \times h$ pixels.

Develop an algorithm, in pseudo-code, that rasterizes the triangle to the screen. The algorithm should be asymptotically optimal in the sense that the processing cost for a triangle are $\mathcal{O}(k)$ if the triangle has k fragments within the screen area (in other words, do not generate fragments outside the screen, or simply reject them after generation; this could have arbitrarily high run-times!).

Your solution does not need to make specific optimizations (integer arithmetic, incremental calculations etc.).

Hint: We did not talk about this in the lecture yet – so be creative! Any correct solution with $\mathcal{O}(k)$ runtime is acceptable.

Solution

A very simple solution could look like that:

Algorithm 1: Parallelogram rasterizer – rasterizes parallelograms with top and bottom side parallel to the x-axis, and arbitrary points within the screen boundaries

Input: Points p_upperLeft, p_upperRight, p_lowerLeft, p_lowerRight $\in \mathbb{R}^2$

Output: list of fragments (would write to framebuffer in real implementation).

```
Assert(p_upperLeft.y = p_upperRight.y);
Assert(p_lowerLeft.y = p_lowerRight.y);
Assert(p_lowerLeft.y >= p_upperLeft.y);
Assert(all inputs within screen coordinates, [0…w-1] × [0…h]
for (int y=p_upperLeft.y; y <= p_lowerRight.y; y++) {
    int x_start = round ( (p_upperLeft.x – p_LowerLeft.x)
                  / (p_upperLeft.y – p_lowerLeft.y) * (y - p_upperLeft.y) );
    int x_end = round ( (p_upperRight.x – p_LowerRight.x)
                  / (p_upperRight.y – p_lowerRight.y) * (y - p_upperRight.y) );
    for (int x=x_start; x<x_end; x++) {
        output_fragment(x,y); // write to framebuffer
                              // or call pixelshader and then write the result
    }
}
```

Remark: some details are not fully worked out below – work out the details yourselves!

Algorithm 2: Convert triangle into parallelograms

Input: Points p1,p2,p3

Output: calls Algorithm 1 for output of fragments

Data structure:

parallelogram (upperLeft, upperRight, lowerLeft, lowerRight) (4 points)

```
Sort points p1,p2,p3 by y-coordinate
p2_mid = point on line p1-p3 at y-coordinate p2.y
parallelogram pg1 = (p1, p1, p2, p2_mid)
parallelogram pg2 = (p2, p2_mid, p3, p3)
parallelogram_list pgl1 = clipParallelogram(pg1);
parallelogram_list pgl2 = clipParallelogram(pg2);
if (visible1) parallelogramRasterizer(pg1);
if (visible2) parallelogramRasterizer(pg2);
```

Remark: some details are not fully worked out below – work out the details yourselves!

Algorithm 3: Clip parallelogram against screen

Input: Points p_upperLeft, p_upperRight, p_lowerLeft, p_lowerRight $\in \mathbb{R}^2$

Output: list of parallelograms

```
If (y-coordinates fully above or below screen) {
    return (empty list).
}
If (p_upperLeft.y <0) {
    clip upper boundary of parallelogram at screen top, rewrite points accordingly
}
If (p_upperLeft.y >= h) {
    clip lower boundary of parallelogram at bottom of the screen,
    rewrite points accordingly
}
If (left edge left right of screen or right edge left of screen) {
    return (empty list);
} otherwise {
    if (left edge intersects screen) {
        split parallelogram into two separate ones at the intersection y-coordinate
        store in result list
    } otherwise {
        store original parallelogram in result list
    }
    for (all parallelograms in result list)
    if (right edge intersects screen) {
        remove from result list
        split parallelogram into two separate ones at the intersection y-coordinate
        store the two resulting ones in the result list
    }
}
```

Remark:

This algorithm can be made quite fast by precomputing the slopes of the line equations in the inner loop of algorithm one. The only operation that is required in the two inner loops is a single addition. This one can be made very fast with fixed point arithmetics, or, more sophisticated, with a mid-point/Bresenham algorithm (not covered in the lecture so far).

On the other hand, we have GPUs these days; who does still implement such things in software… :-)

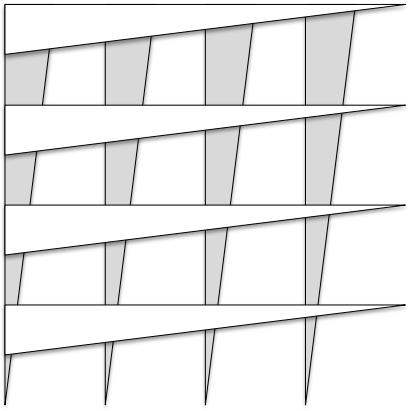
Assignment #2: *No overpaint allowed! – a variant of the painter's algorithm.*

Imagine we need to create a variant of the painter's algorithm where no overwriting is allowed: Given the projection of n triangles into 2D, we have to cut the scene into smaller triangles such that invisible area is never drawn (in other words, if area overlaps with another triangle, it must be removed; the output are still triangles). Such an algorithm might for example be necessary to drive a plotter (a devise that draws wire-frame drawings of a 3D scene and cannot erase anything it has ever drawn).

Prove a quadratic lower-bound for the worst-case complexity. Show that there exists scenes with $\mathcal{O}(n)$ input triangles that create $\mathcal{O}(n^2)$ output triangles.

Hint: One can construct a scene (more specifically, a family of similar scenes with an arbitrary number of triangles) such that the number of required pieces grows quadratically.

Solution



Example scene: *n*/2 horizontal and *n*/2 vertical triangles

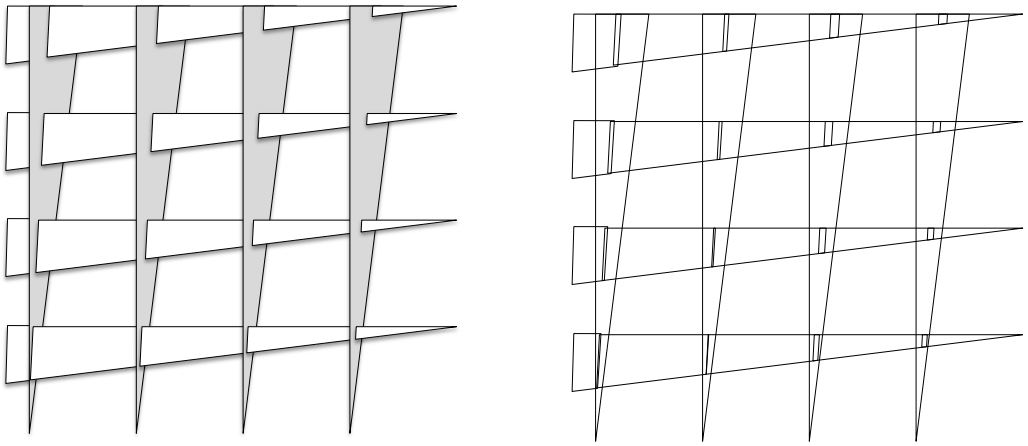
Assignment #3: *The real painter's algorithm, and why it can be slow…*

Now we permit overdraw. Consider again a "correct" painters algorithm that cuts triangles into smaller pieces until the triangles can be brought into a fixed order such that painting them in that order makes sure that the parts always overwrite parts that are further away.

Prove that such an algorithm has also a quadratic worst-case complexity. More specifically, show that at least one scene exists, consisting of n triangles where $\mathcal{O}(n^2)$ sub-triangle (pieces) must be generated before such an ordering can be obtained.

Hint: One can construct a scene (more specifically, a family of similar scenes with an arbitrary number of triangles) such that the number of required pieces grows quadratically.

Remark: Assignment 3 is more difficult than assignment 2 – the example is not as easy to find as before.



Example scene: *n*/2 horizontal and *n*/2 vertical triangles,
intersecting each other in a stack of increasing depth.
Making this in PPT was actually a lot of work! (painter's solution – right image)

Assignment #4: *Normals and back-face culling*

a) Calculate the unit normal vector of the following triangles! Use the CCW rule – points are given in counter-clockwise direction if viewed from the outside. The normal should point outside (towards the viewer) when looking at the visible side.

$$t_1 = (\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$$

$$\mathbf{p}_1 = \begin{pmatrix}0\\0\\1\end{pmatrix}, \quad \mathbf{p}_2 = \begin{pmatrix}1\\0\\1\end{pmatrix}, \quad \mathbf{p}_3 = \begin{pmatrix}1\\2\\1\end{pmatrix}$$

$$t_2 = (\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3)$$

$$\mathbf{q}_1 = \begin{pmatrix}0\\0\\2\end{pmatrix}, \quad \mathbf{p}_2 = \begin{pmatrix}5\\5\\0\end{pmatrix}, \quad \mathbf{p}_3 = \begin{pmatrix}7\\2\\-1\end{pmatrix}$$

Solution: t_1: $(0,0,-1)^T$.

Solution: t_2: use cross product of sides to compute. Orient points according to CCW for ordering the input to the cross-product.

b) Our rendering system implements back-face culling – triangles for which the normal vector (orthogonal to the triangle plane). The camera is located in the origin and looking down the positive z-axis. Which of the two following triangles is visible after CCW-back-face culling?

$$t_1 = (\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$$

$$\mathbf{p}_1 = \begin{pmatrix}0\\0\\1\end{pmatrix}, \quad \mathbf{p}_2 = \begin{pmatrix}1\\0\\1\end{pmatrix}, \quad \mathbf{p}_3 = \begin{pmatrix}1\\1\\1\end{pmatrix}$$

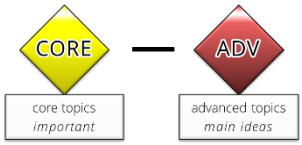
$$t_2 = (\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3)$$

$$\mathbf{q}_1 = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}, \qquad \mathbf{p}_2 = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}, \qquad \mathbf{p}_3 = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

Explain briefly why.

Solution: Only the first one. Because it is oriented CCW.

Assignment #5: *Diffuse shading*

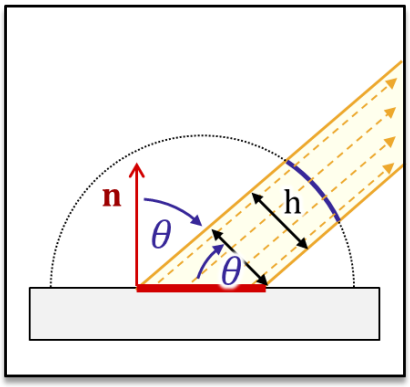


Consider the diffuse ("Lambertian") shading model. Explain why the brightness of the surfaces decreases with $\cos\theta$, where θ is the angle between the incident light direction $\mathbf{l}$ and the surface normal $\mathbf{n}$.

Hint: Consider a parallel ray bundle that hits a small piece of surface. Then vary the incident angle θ and describe what happens (does not need to be formally strict; formally strict would be very much rated "red").

Solution:

Consider the following image:



Assuming that each light ray has constant power density, the overall power arriving is proportional to the height h of the yellow ray bundle. This height is proportional to $\cos\theta$.

Remark: A strictly formal argument involves defining radiance (power density of rays) as infinitesimal density quantity with respect to solid angle and area. Then, by integrating over the finite area (bold red) and considering a suitable directional light source (e.g., a point light source in the limit of moving farer away and scaling up in brightness by dist² to compensate), we can get the same result with physically well-defined quantities. The required radiometric quantities such as radiance have not been formally defined in this lecture; therefore, this is for now going too far; the intuition above should be sufficient.