UU Graphics

academic year 2013/14 – 4th period

Theoretical Assignment #6:
Rasterization, Shading, and Raytracing

June 17 2014

Assignment #1: *2D Texture Mapping*

In this assignment, we want to derive a 2D texture mapping algorithm. The idea is to find an affine map that maps pixels on a 2D triangle to texels in a 2D texture.

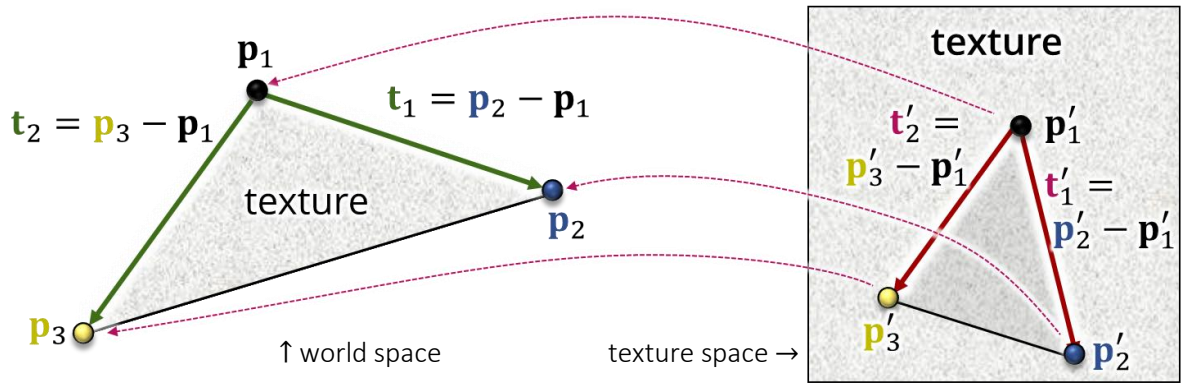


Figure: the situation in this assignment

You are given the following information (see figure above):

Input: A 2D triangle with points (in "world-space") $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3 \in \mathbb{R}^2$. Each of the points has a texture coordinate $\mathbf{p}_1', \mathbf{p}_2', \mathbf{p}_3' \in \mathbb{R}^2$ ("texture-space") that describe where the triangle would be mapped within the texture space.
Both the triangle in world space and in texture space is non-degenerate, i.e., the points form a proper triangle and are not all located on a single line. In other words, the side vectors $\mathbf{t}_1, \mathbf{t}_2$ shown in the figure above are linearly independent, and the same holds for $\mathbf{t}_1', \mathbf{t}_2'$.

We know want to study this mapping further.

a) The specification above determines exactly one affine map between the world space and the texture space (in both directions, i.e., including an inverse mapping). Why?
Remark: An affine map is a combination of a linear map and a translation, i.e., of the form

$$f(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{t}, \qquad \mathbf{x}, \mathbf{t} \in \mathbb{R}^2, \qquad \mathbf{M} \in \mathbb{R}^{2\times 2}.$$

(You can also write this in homogeneous coordinates, as a 3×3 matrix, if you like this better.)

Solution:

We map from one coordinate system to another. Aligning the origins accounts for a translation, in addition we have to map two linearly independent vectors to two other such vectors. This uniquely defines a linear map (because we prescribe the images of basis vectors for $\mathbb{R}^2$). The mapping is unique (the vectors form a basis) because they are linearly independent. This is also the reason why the map is invertible.

b) How can we compute the map from world space to texture space, i.e., determine a matrix $\mathbf{M}$ and a translation vector $\mathbf{t}$ (or an equivalent homogeneous matrix) that determines for each pixel $\mathbf{x} \in \mathbb{R}^2$ in the image the corresponding texture coordinate in texture space (also $\mathbb{R}^2$).

Solution:

The mapping is obtained as

$$\mathbf{x} \to \mathbf{p}_1' + \begin{pmatrix} | & | \\ \mathbf{t}_1' & \mathbf{t}_2' \\ | & | \end{pmatrix} \cdot \begin{pmatrix} | & | \\ \boldsymbol{t}_1 & \boldsymbol{t}_2 \\ | & | \end{pmatrix}^{-1} (\mathbf{x} - \mathbf{p}_1).$$

We first move the origin of the triangle coordinate system to $\mathbf{p}_1$, then we convert the world coordinates into triangle coordinates (rightmost matrix). These are then mapped to texture space (leftmost matrix) and the origin is adjusted again. Written as single affine map, we obtain:

$$\mathbf{x} \to \mathbf{M}\mathbf{x} + (\mathbf{p}_1' - \mathbf{M}\mathbf{p}_1)$$
$$\text{with } \mathbf{M} := \begin{pmatrix} | & | \\ \mathbf{t}_1' & \mathbf{t}_2' \\ | & | \end{pmatrix} \cdot \begin{pmatrix} | & | \\ \boldsymbol{t}_1 & \boldsymbol{t}_2 \\ | & | \end{pmatrix}^{-1}$$

c) For rasterization, it is important to have cheap inner loops. In this case, when moving from pixel $\mathbf{x} = (x, y)^{\mathrm{T}}$ to pixel $\mathbf{x}_{oneRight} = (x + 1, y)^{\mathrm{T}}$, i.e., just one pixel to the right, we can update the texture coordinate for the previous pixel by just adding an increment vector to the last known texture coordinates.
How can we compute this increment vector?
Hint: Examine how the affine map $f(\mathbf{x})$ changes if you move from $\mathbf{x}$ to $\mathbf{x} + (1,0)^{\mathrm{T}}$.

Solution:

Fortunately, everything is linear/affine (1st order); so all directional derivatives of the map are constants. We can see the effect without using multi-dimensional derivatives, by directly plugging in the desired steps: Let $\mathbf{r}$ be the step vector, then:

$$f(\mathbf{x} + \mathbf{r}) - f(\mathbf{x}) = [\mathbf{M}(\mathbf{x} + \mathbf{r}) + \mathbf{t}] - [\mathbf{M}\mathbf{x} + \mathbf{t}]$$
$$= \mathbf{M}\mathbf{x} + \mathbf{M}\mathbf{r} + \mathbf{t} - \mathbf{M}\mathbf{x} - \mathbf{t}$$
$$= \mathbf{M}\mathbf{r}$$

Hence, the difference between $f(\mathbf{x}+\mathbf{r})$ and $f(\mathbf{x})$ is $\mathbf{Mr}$. For the case of stepping one pixel to the right, the result is $\mathbf{M}\binom{1}{0}$, i.e., the first column of matrix $\mathbf{M}$ as defined in (b).

d) To make it really "fancy": The outer loop of the rasterizer runs along an edge of the triangle (starting at a projected vertex). How can we update the texture coordinate when we follow a general line in world coordinates (such as the edge of a triangle)?
Hint: Examine how the affine map $f(\mathbf{x})$ changes if you move from $\mathbf{x}$ to $\mathbf{x}+\mathbf{r}$ for a vector $\mathbf{r} \in \mathbb{R}^2$.

Solution: see part (c)

Assignment #2: *Shooting rays!*

In this assignment, we look at how to create specific rays in a raytracing algorithm. We will use this in next week's lecture to build a (recursive) raytracer.

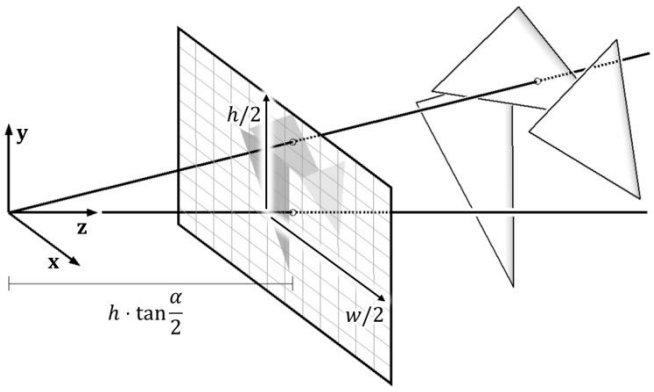
a) You are given a perspective camera with vertical view angle α, and a screen of $w \times h$ square pixels. The camera coordinate system is located in the origin of the coordinate frame, viewing along the positive z-axis, and x-, and y-axis being the x-, and y-axis of the virtual screen.

Determine a parametric ray equation for a ray that starts in the origin and crosses through pixel $(i,j) \in \{1,\dots,w\} \times \{1,\dots,h\}$.

These rays are called *primary rays*, and they are traced first when using raytracing for rendering (as discussed in the section on visibility algorithms).

Solution:

First, here is an illustration:



The code below creates all primary rays; a single ray can be obtained by dropping the two nested loops and executing the code for fixed x=i, y=j.

Generating Primary rays

```
float tanOfHalfViewingAngle = tan(vf.getVerticalFieldOfView()/180.0f*M_PI/2.0f);
Vector3f up = cam.getOrthoNormUpDirection() * tanOfHalfViewingAngle;
Vector3f right = cam.getOrthoNormRightDirection() * tanOfHalfViewingAngle;
Vector3f view = cam.getOrthoNormViewDirection();
Ray3f ray;
ray.origin = cam.getPosition();
float aspectRatio = (float32)width/(float32)height;
for (int y=0; y<height; y++) {
   float rely = 2.0f * (height/2 - y) / (float)height;
   for (int x=0; x<width; x++) {
      float relx = 2.0f * (width/2 - x) / (float)width * aspectRatio;
       ray.direction = view + right * relx + up * rely;
       tracePrimaryRay(ray);
       … process result…
    }
}
```

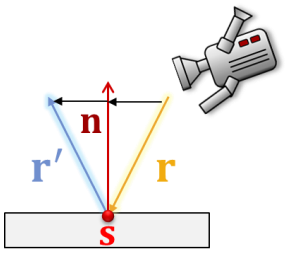
b) We can also have secondary rays that are continuations of the tracing when hitting a reflective or transparent surface. In this context, the following problem comes up:

   You are given a ray $\{\lambda \cdot \mathbf{r} + \mathbf{p} | \lambda \geq 0\}$, with direction vector $\mathbf{r} \in \mathbb{R}^3$ and starting point $\mathbf{p} \in \mathbb{R}^3$. The ray hits a surface point $\mathbf{s} \in \mathbb{R}^3$ which has a surface normal of $\mathbf{n} \in \mathbb{R}^3$ (at some parameter λ_0, which does not matter for the computation).

   First, draw this situation in a 2D sketch. Then, compute a parametric equation for the reflected ray (it will dependent on $\mathbf{r}, \mathbf{n}, \mathbf{s}$).

Solution:

In analogy to the lecture slides (on computing reflections), we obtain the reflected direction as:



$$\mathbf{r}' = 2(\mathbf{r} - \langle \mathbf{n}, \mathbf{r} \rangle \cdot \mathbf{n}) - \mathbf{r}, \text{ assuming } \|\mathbf{n}\| = 1$$

The new ray is then given by the set $\{\lambda \cdot \mathbf{r}' + \mathbf{s} | \lambda \geq 0\}$ (which also contains the parametric representation with constraints on the parameter).

Assignment #3: *Clipping*

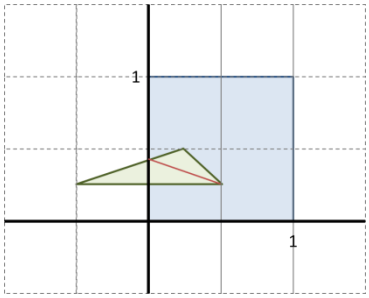
Now we want to clip a triangle against the rectangular bounding box of the screen.

a) A simple 2D example.

Assume that your screen is a unit square, with (floating-point) coordinates $[0,1] \times [0,1]$. Consider the triangle spanned by the three points

$$\mathbf{p}_1 = (-0.5, 0.25)^{\mathrm{T}}, \qquad \mathbf{p}_2 = (0.5, 0.25)^{\mathrm{T}}, \qquad \mathbf{p}_3 = (0.25, 0.5)^{\mathrm{T}}$$

Clip this triangle manually against the screen! Split it into multiple triangles, if required. Draw the situation first to avoid unnecessary computations.



The edges $\{\mathbf{p}_1, \mathbf{p}_2\}$ and The edges $\{\mathbf{p}_1, \mathbf{p}_3\}$ are intersecting the y-axis. We have to setup the line equations and compute the value at **x** = 0, which yields two new points

$$\mathbf{p}_4 = \left(0, \frac{1}{4} + \frac{3}{2} \cdot \frac{1}{4}\right)^{\mathrm{T}} = \left(0, \frac{5}{8}\right)^{\mathrm{T}}, \qquad \mathbf{p}_5 = (0, 0.25)^{\mathrm{T}}$$

We then form two new triangles in the visible part (red edge in figure above).

b) And now for 3D!

We now want to clip a triangle against a half space

$$H_{n,d} = \{\mathbf{x} | \mathbf{x} \in \mathbb{R}^3 \text{ and } \langle \mathbf{n}, \mathbf{x} \rangle \geq d\}.$$

$\mathbf{n} \in \mathbb{R}^3$ and $d \in \mathbb{R}$ are the parameters of an implicit plane equation. A half-space $H_{n,d}$ is the space on the "outside" a plane, if we consider the normal of the plane pointing outwards, as defined above (in this case, the scalar product is positive; the constant d can shift the plane outside the origin).

For simplicity, we only consider culling a single triangle edge $(\mathbf{p}_1, \mathbf{p}_2)$, with $\mathbf{p}_1, \mathbf{p}_2 \in \mathbb{R}^3$ against $H_{n,d}$, i.e., we only want to retain the part inside the half-space.

Describe an algorithm for computing the segment inside the half space. The algorithm should return "empty set" as answer if the edge is completely outside. You can assume that the edge is not parallel to the plane.

Solution:

We have a parametric equation with constraints

$$\mathbf{x}(t) = \mathbf{p}_1 + (\mathbf{p}_2 - \mathbf{p}_1)t, t \in [0,1]$$

for the edge. We first intersect the line of the edge with the plane of the half space:

$$\begin{aligned}
&\langle \mathbf{n}, \mathbf{x} \rangle = d \wedge \mathbf{x}(t) = \mathbf{p}_1 + (\mathbf{p}_2 - \mathbf{p}_1)t \\
&\Leftrightarrow \langle \mathbf{n}, \mathbf{p}_1 + (\mathbf{p}_2 - \mathbf{p}_1)t \rangle - d = 0 \\
&\Leftrightarrow \langle \mathbf{n}, \mathbf{p}_1 \rangle + \langle \mathbf{n}, (\mathbf{p}_2 - \mathbf{p}_1) \rangle t - d = 0 \\
&\Leftrightarrow t = d - \frac{\langle \mathbf{n}, \mathbf{p}_1 \rangle}{\langle \mathbf{n}, (\mathbf{p}_2 - \mathbf{p}_1) \rangle}
\end{aligned}$$

Afterwards, we check for four case:

- Both t are between [0,1]: keep edge as is
- One is outside, one inside: replace outside point with x(0) or x(1) (depending on wether it is too large or to small).
- Both are outside: return empty set.