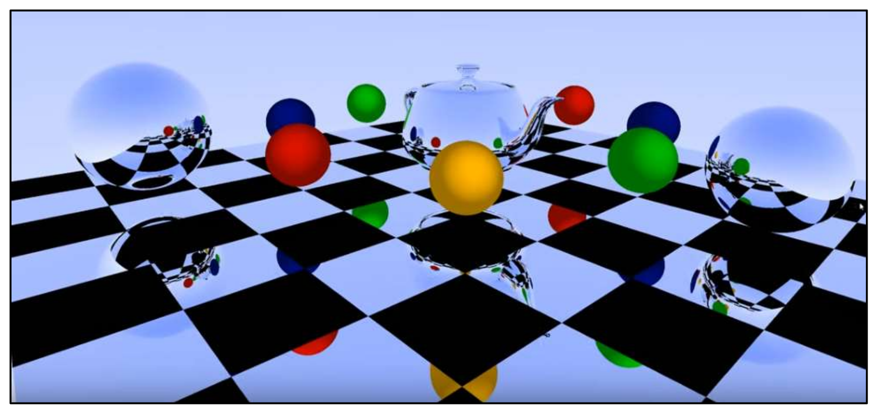
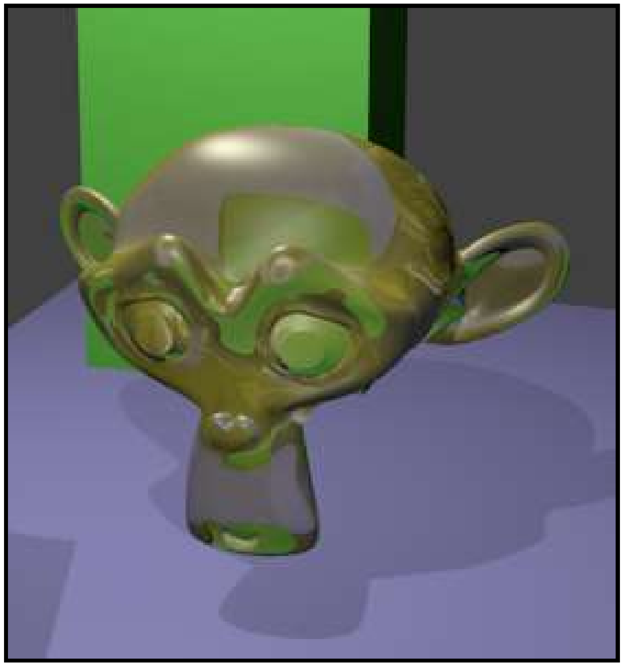
Vulkan Ray Tracing – 5 New Shader Types!

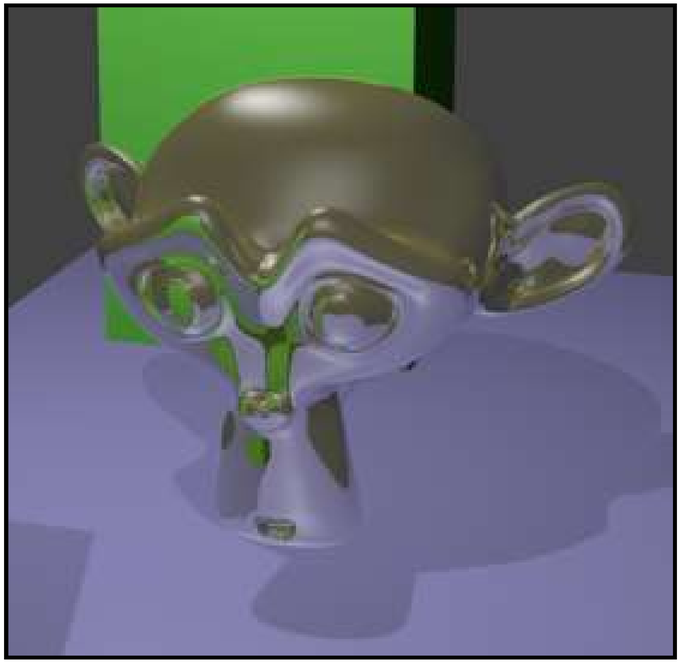
Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu



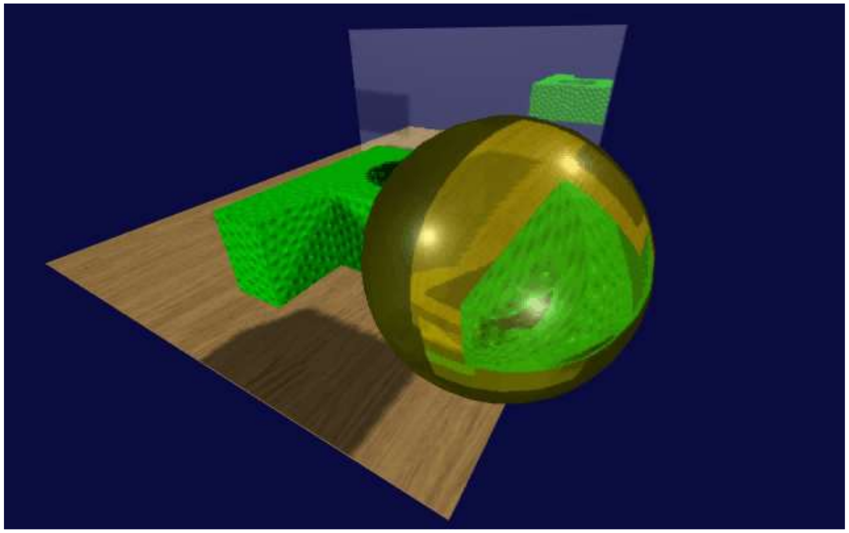
Analog Ray Tracing Example ☺

Digital Ray Tracing Examples



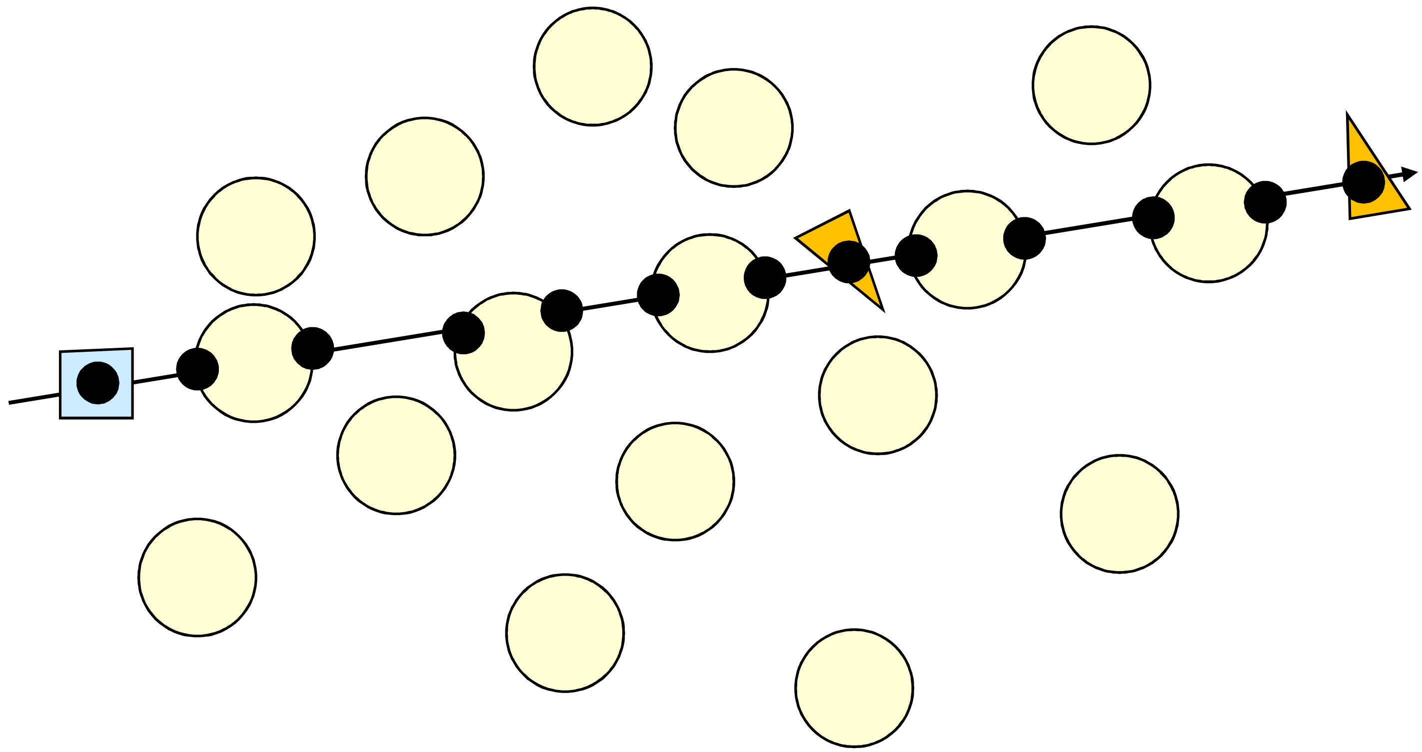


Blender



IronCad

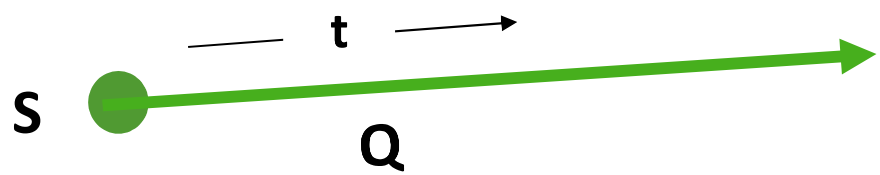
In a Raytracing, each ray typically hits a lot of Things



Parametrizing a Ray

Given:
S is the (x,y,z) starting point
Q is the (x,y,z) direction of travel

Then, the (x,y,z) position of a point ***p*** at some position along its direction of travel is:



$$p = S + tQ$$

$$t \geq 0.$$

Example: The Ray Intersection Process for a Sphere

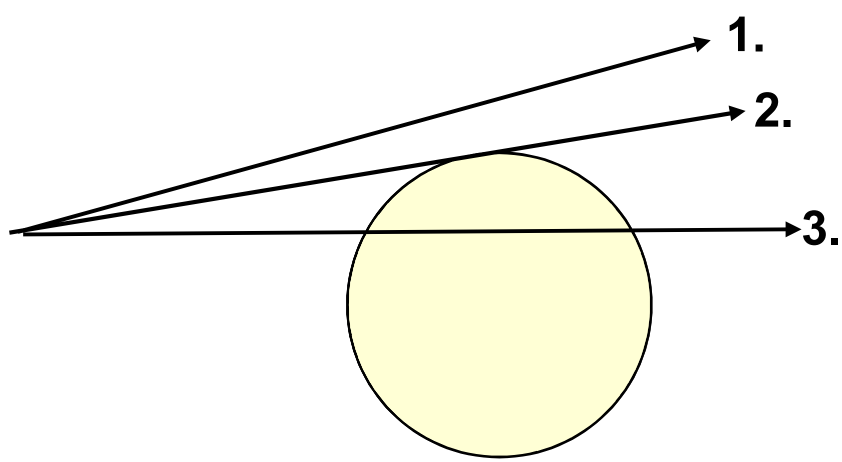
Sphere equation: $(x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 = R^2$
Ray equation: $(x,y,z) = (x_0,y_0,z_0) + t*(dx,dy,dz)$

Plugging (x,y,z) from the second equation into the first equation and multiplying-through and simplifying gives:

$$At^2 + Bt + C = 0 \quad \Rightarrow \quad t_1, t_2 = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

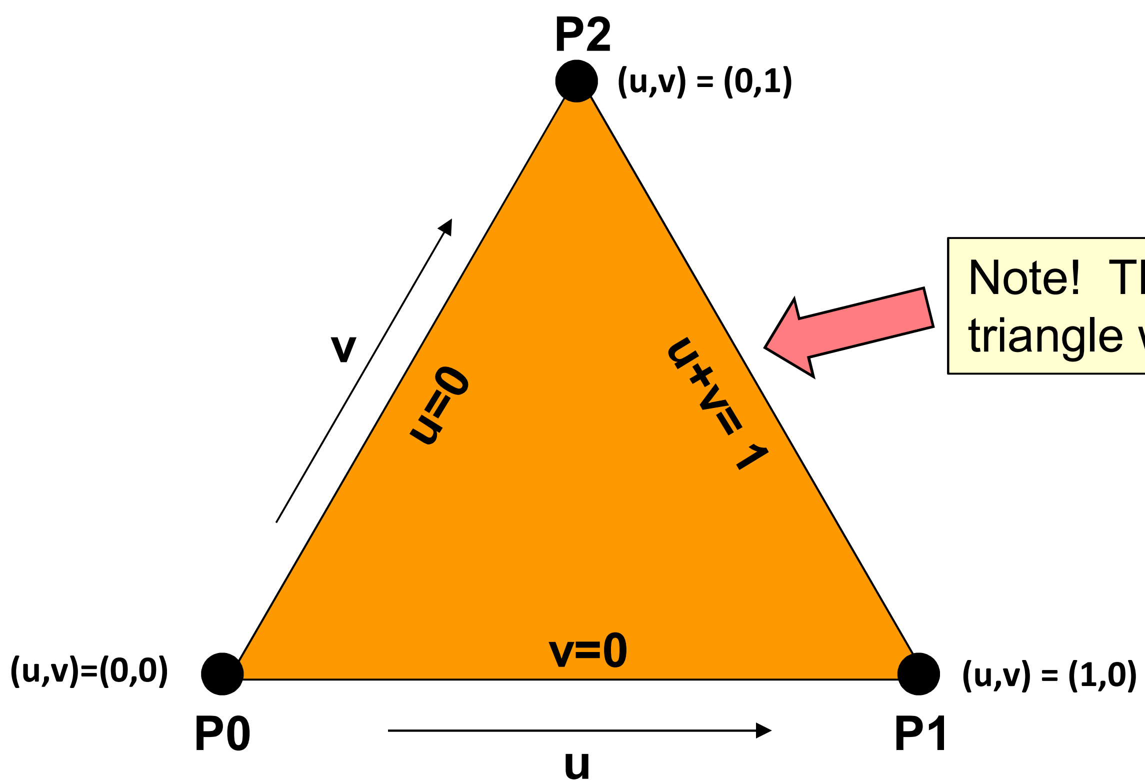
Solve for t_1, t_2 and analyze the solution like this:

1. If both t_1 and t_2 are complex (i.e., have an imaginary component), then the ray missed the sphere completely.
2. If both t_1 and t_2 are real and identical, then the ray brushed the sphere at a tangent point.
3. If both t_1 and t_2 are real and different, then the ray entered and exited the sphere.



Parameterizing a Triangle

It’s often useful to be able to parameterize a triangle into (u,v), like this:



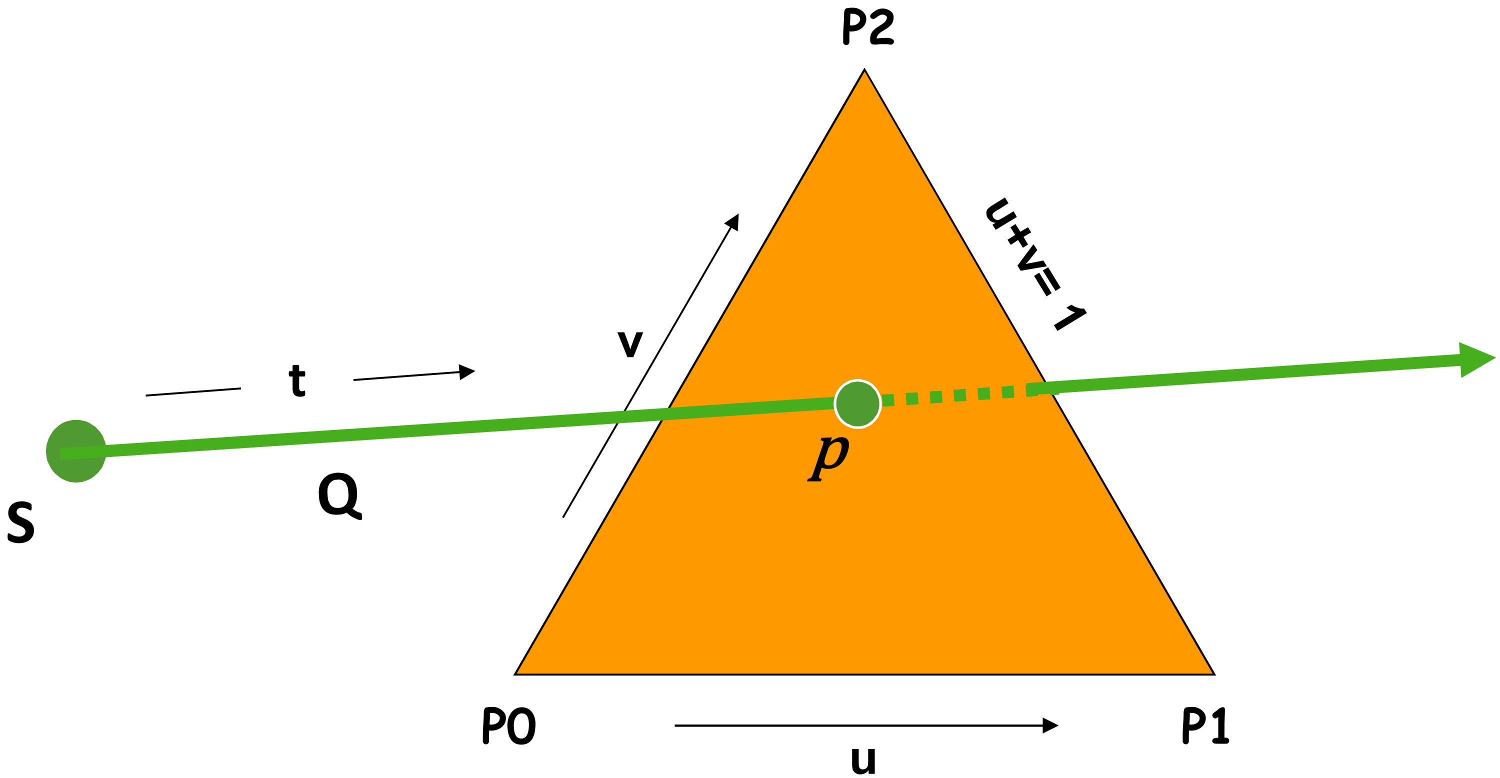
Note! There is *no* place in this triangle where u = 1 and v = 1.

$$p = P0 + u*(P1-P0) + v*(P2-P0)$$

The Setup

We want to find out where the ray intersects the triangle.

That is, where is the point p that is common to both the ray and the triangle?



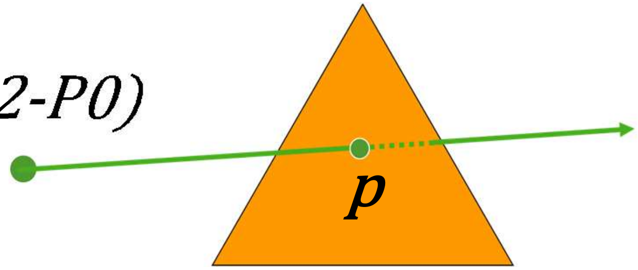
Such that:

$$t \geq 0.$$
$$0. \leq u \leq 1.$$
$$0. \leq v \leq 1.\text{-u}$$

Equation Setup

Triangle: $p = P0 + u*(P1\text{-}P0) + v*(P2\text{-}P0)$

Ray: $p = S + tQ$



Re-arranging:

$$P0 + u*(P1\text{-}P0) + v*(P2\text{-}P0) = S + tQ$$

Re-arranging some more:

$$-tQ + u*(P1\text{-}P0) + v*(P2\text{-}P0) = S - P0$$

Then collecting terms, we get:

$$At + Bu + Cv = D$$

where:

$$A = \text{-}Q$$

$$B = P1\text{-}P0$$

$$C = P2\text{-}P0$$

$$D = S - P0$$

Three Equations, Three Unknowns

Remembering that this equation is really 3 equations in (x,y,z):

$$At + Bu + Cv = D$$

we have 3 equations with 3 unknowns, which can be cast into a matrix form

$$\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix} \begin{Bmatrix} t \\ u \\ v \end{Bmatrix} = \begin{Bmatrix} D_x \\ D_y \\ D_z \end{Bmatrix}$$

Our goal is to solve this for t*, u*, and v*

Solve for (t*,u*,v*) using Cramer's Rule

$$\begin{bmatrix} A_x & B_x & C_x \\ A_z & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix} \begin{Bmatrix} t \\ u \\ v \end{Bmatrix} = \begin{Bmatrix} D_x \\ D_y \\ D_z \end{Bmatrix}$$

$$D_0 = det \begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix}$$

$$D_t = det \begin{bmatrix} D_x & B_x & C_x \\ D_y & B_y & C_y \\ D_z & B_z & C_z \end{bmatrix} \qquad \boldsymbol{t^* = \frac{D_t}{D_0}}$$

$$D_u = det \begin{bmatrix} A_x & D_x & C_x \\ A_y & D_y & C_y \\ A_z & D_z & C_z \end{bmatrix} \qquad \boldsymbol{u^* = \frac{D_u}{D_0}}$$

$$D_v = det \begin{bmatrix} A_x & B_x & D_x \\ A_y & B_y & D_y \\ A_z & B_z & D_z \end{bmatrix} \qquad \boldsymbol{v^* = \frac{D_v}{D_0}}$$

The Steps

1. Compute D_0
2. If $D_0 \approx 0.$, then the ray is *parallel* to the plane of the triangle

3. Compute D_t
4. Compute t^*
5. If $t^* < 0.$, the ray goes away from the triangle

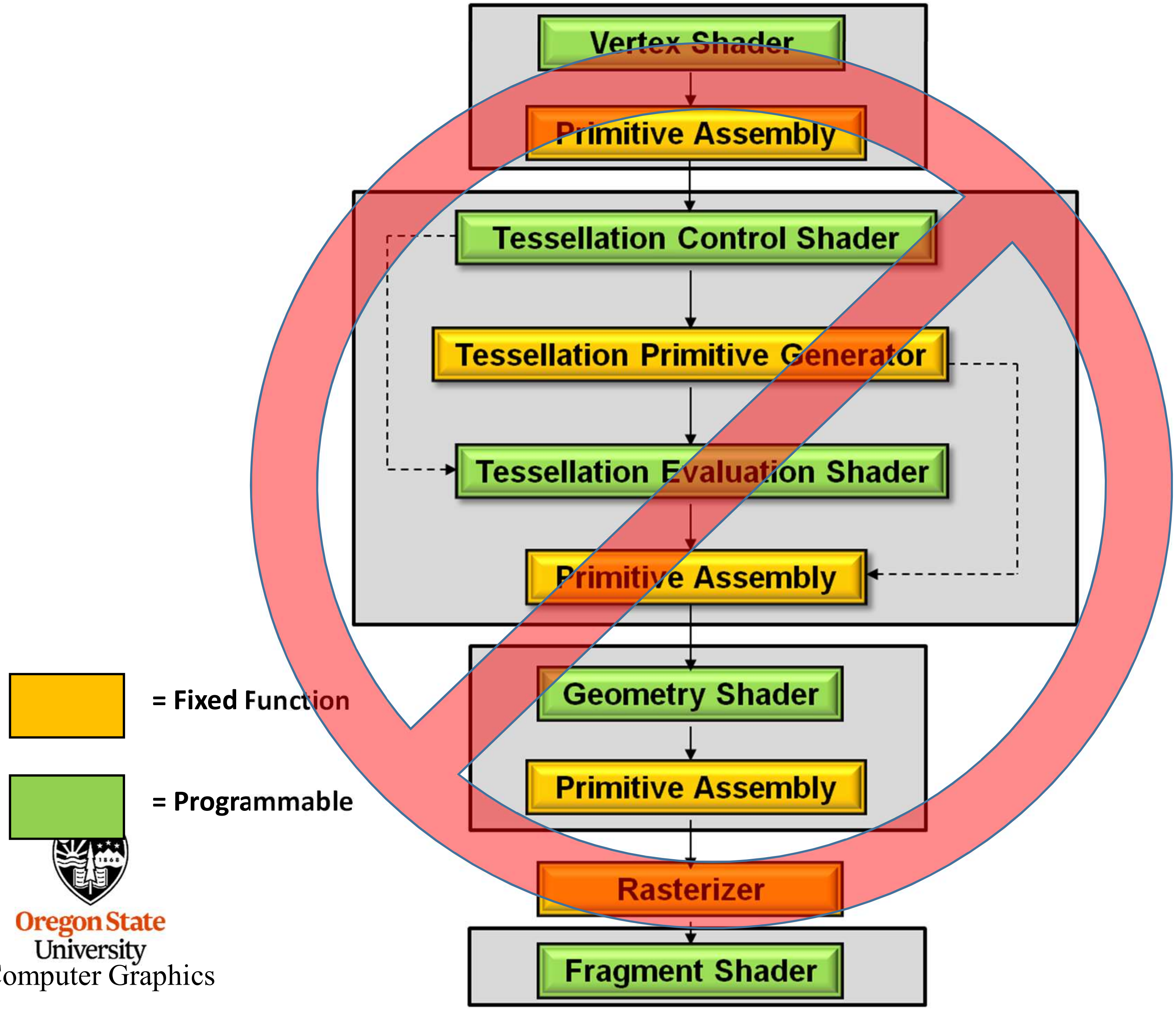
6. Compute D_u
7. Compute u^*
8. If $u^* < 0.$ or $u^* > 1.$, then the ray hits outside the triangle

9. Compute D_v
10. Compute v^*
11. If $v^* < 0.$ or $v^* > 1.-u^*$, then the ray hits outside the triangle

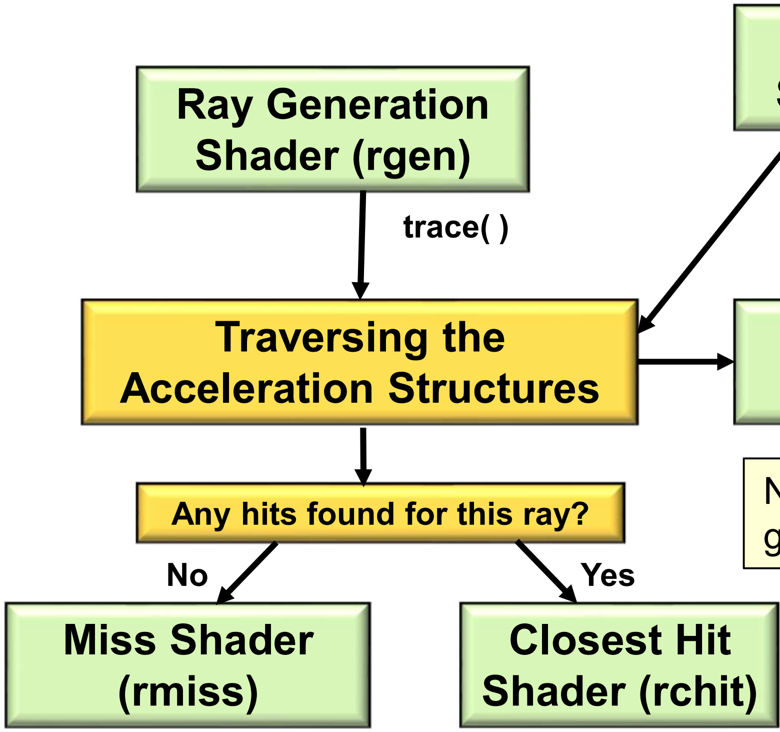
12. The intersection is at the point $\boldsymbol{p = S + Qt^*}$

This is known as the **Möller-Trumbore Triangle Intersection Algorithm**

The Rasterization Shader Pipeline That You Are used to Doesn't Apply to Vulkan Ray Tracing



The Vulkan Ray Tracing Pipeline Involves Five New Shader Types



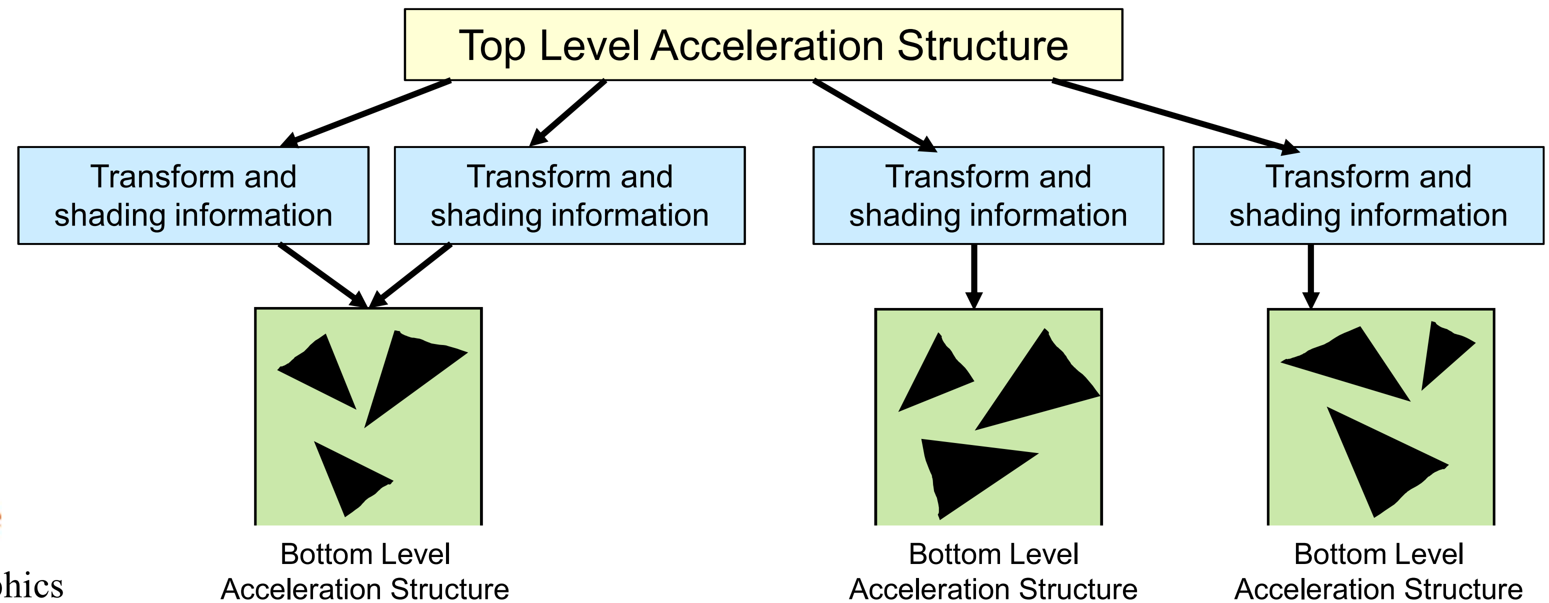
Unlike the rasterization pipeline, there is no constant flow from one shader to the next. Rather, particular shaders are called to respond to particular ***events***.

Note: none of this lives in the hardware meant for rasterization graphics. This is all built on top of the GPU *compute* functionality.

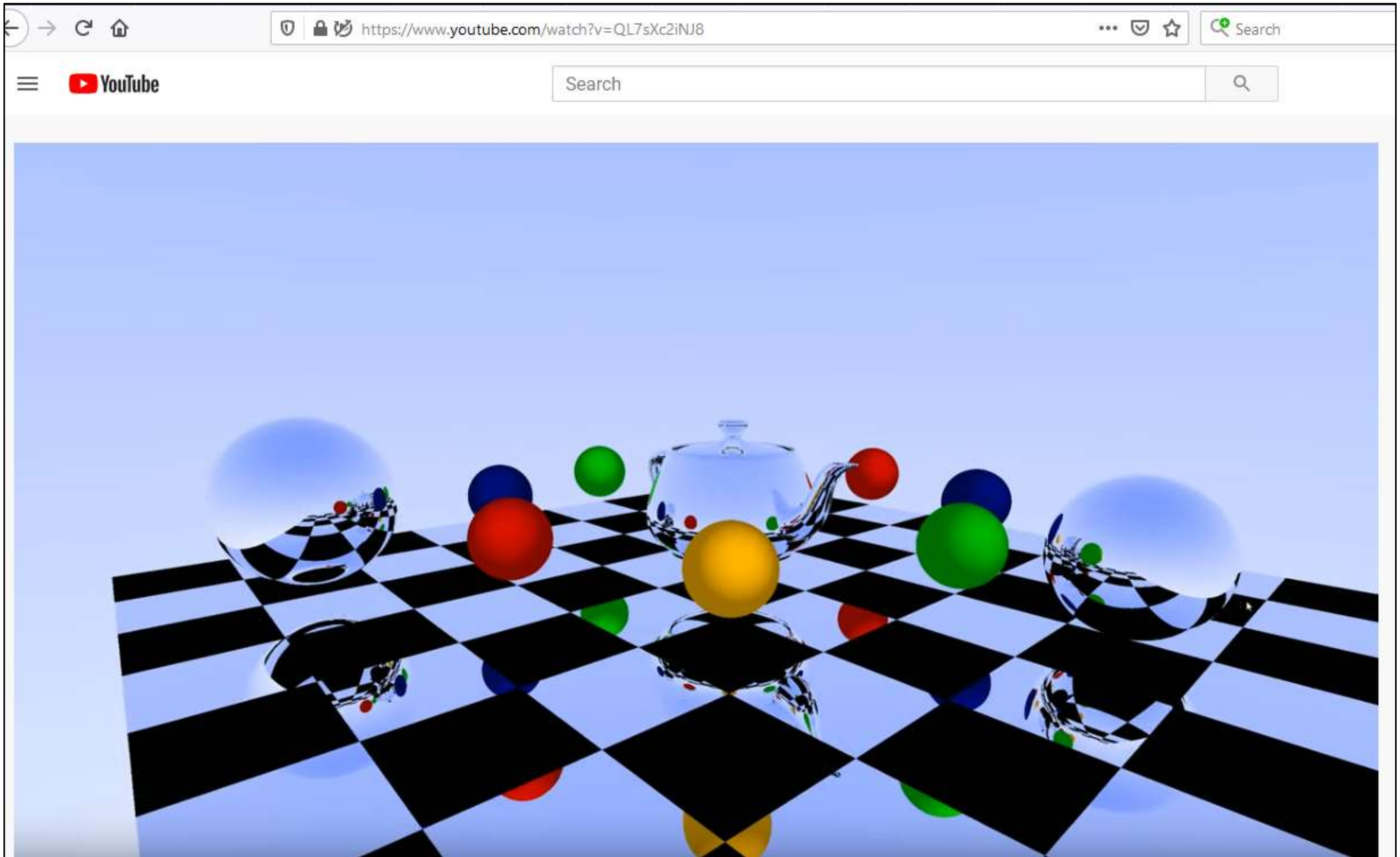
- A **Ray Generation Shader** runs on a 2D grid of threads. It begins the entire ray-tracing operation.
- An **Intersection Shader** implements ray-primitive intersections.
- An **Any Hit Shader** is called when the Intersection Shader finds a hit. It decides if that intersection should be accepted or ignored.
- The **Closest Hit Shader** is called with the information about the hit that happened closest to the viewer. Typically, lighting is done here, or firing off new rays to handle shadows, reflections, and refractions.
- A **Miss Shader** is called when no intersections are found for a given ray. Typically, it just sets its pixel color to the background color.

Acceleration Structures

- A Bottom-level Acceleration Structure (BLAS) reads the vertex data from vertex and index VkBuffers to determine bounding boxes.
- You can also supply your own bounding box information to a BLAS.
- A Top-level Acceleration Structure (TLAS) holds transformations and pointers to multiple BLASes.
- The BLAS is essentially used as a Model Coordinate bounding box, while the TLAS is used as a World Coordinate bounding box.



Check This Out!



https://www.youtube.com/watch?v=QL7sXc2iNJ8