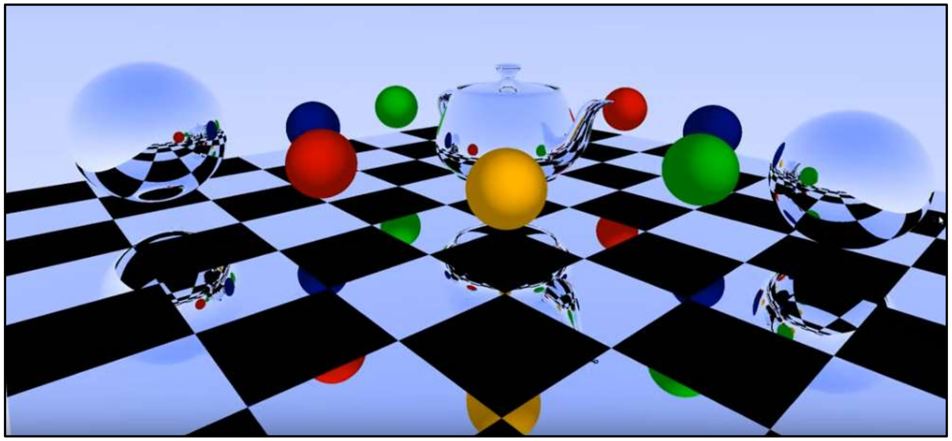
Vulkan Ray Tracing – 5 New Shader Types!

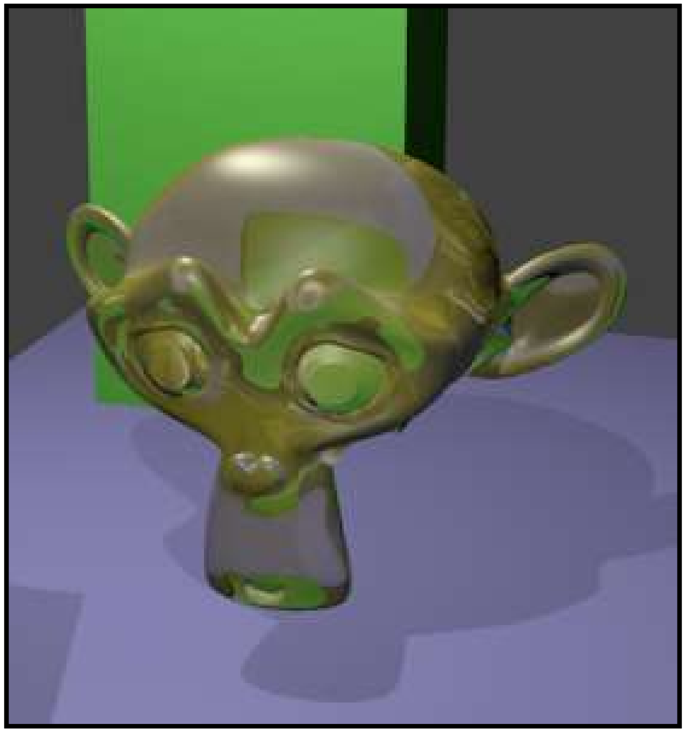
Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

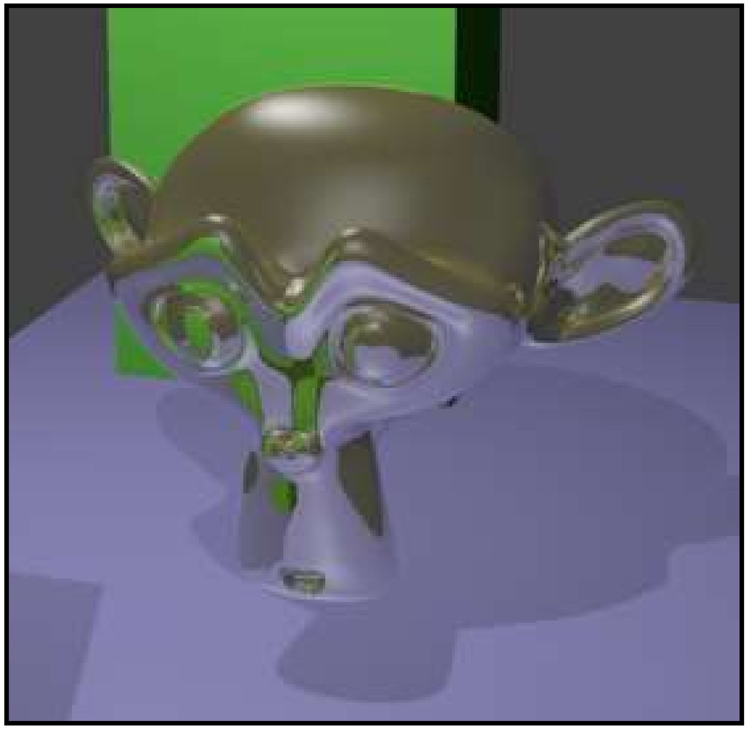
This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License



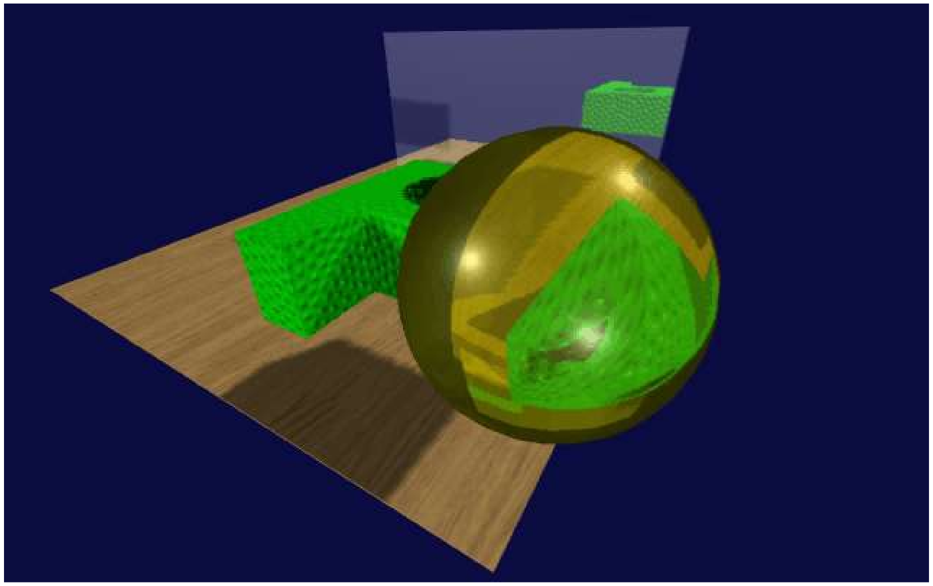
Analog Ray Tracing Example ☺

Digital Ray Tracing Examples



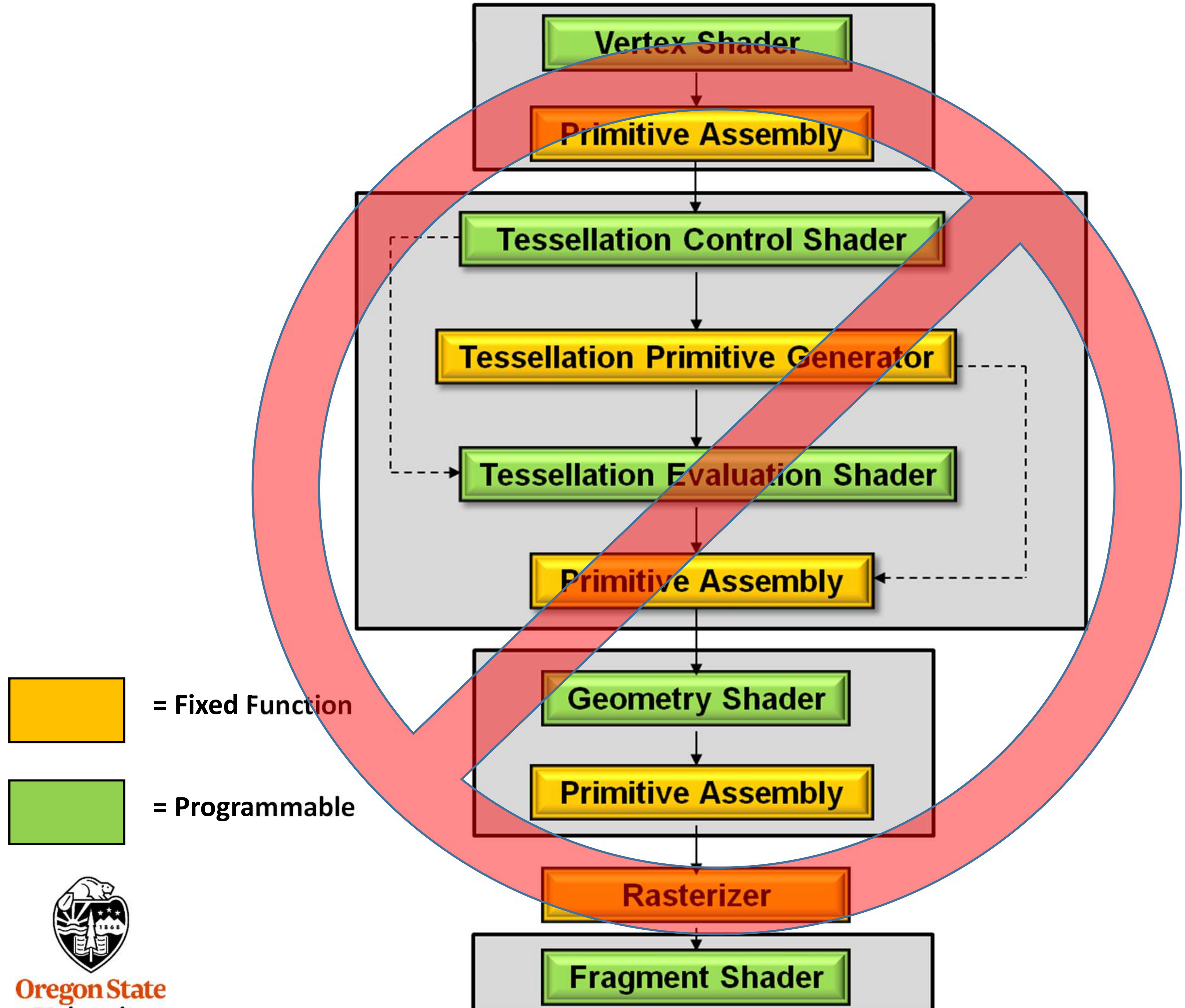


Blender



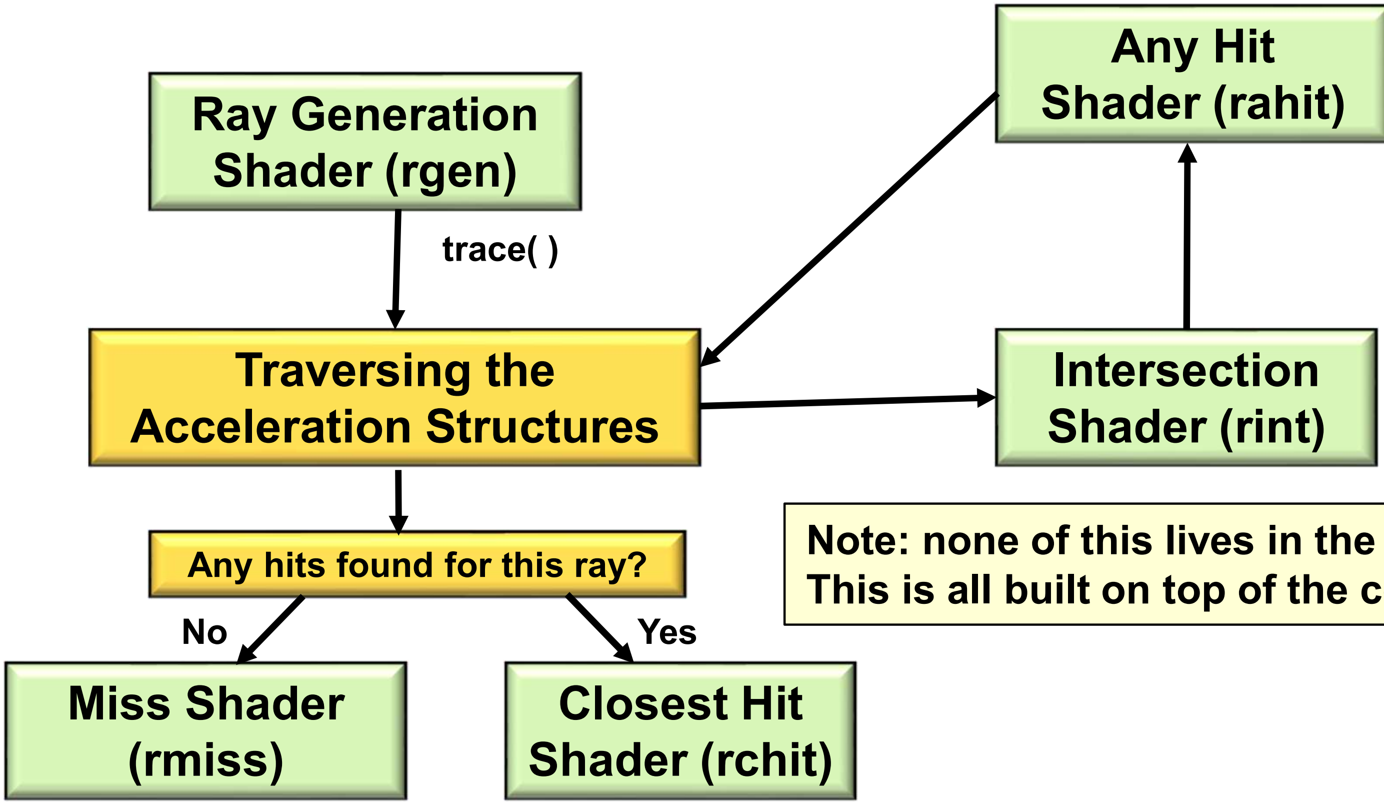
IronCad

The Rasterization Shader Pipeline Doesn't Apply to Ray Tracing



Oregon State University
Computer Graphics

The Ray-trace Pipeline Involves Five New Shader Types



Note: none of this lives in the graphics hardware pipeline. This is all built on top of the 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.
- 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 reflection and refraction.
- 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.

The Ray Intersection Process for a Sphere

1. Sphere equation: $(x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 = R^2$

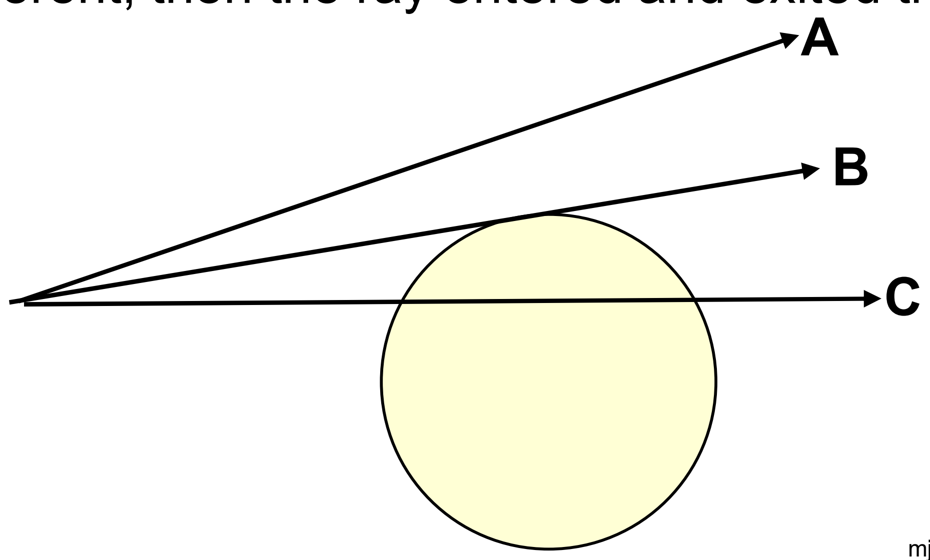
2. Ray equation: $(x,y,z) = (x_0,y_0,z_0) + t^*(dx,dy,dz)$

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

$At^2 + Bt + C = 0$

Solve for t_1, t_2

A. If both t_1 and t_2 are complex, then the ray missed the sphere.
B. If $t_1 == t_2$, then the ray brushed the sphere at a tangent point.
C. If both t_1 and t_2 are real and different, then the ray entered and exited the sphere.



In Vulkan terms:
gl_WorldRayOrigin = (x_0,y_0,z_0)
gl_Hit = t
gl_WorldRayDirection = (dx,dy,dz)

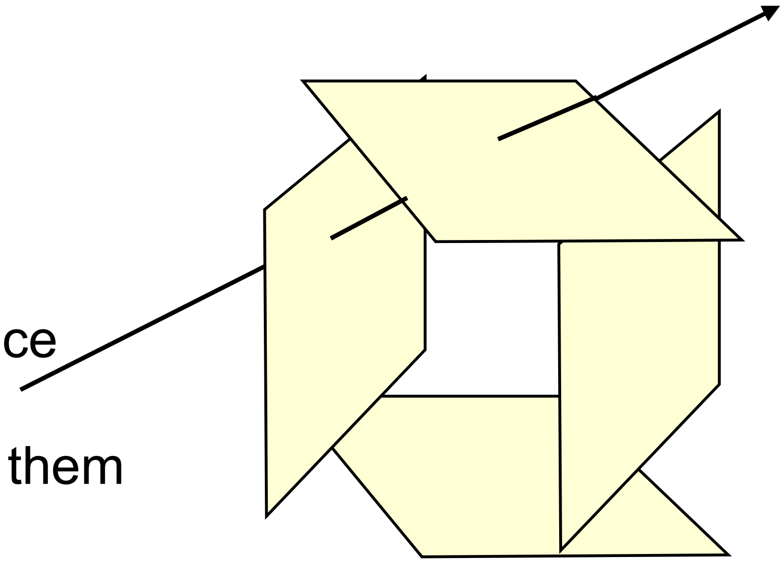
The Ray Intersection Process for a Cube

1. Plane equation: $Ax + By + Cz + D = 0$

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:

$At + B = 0$
Solve for t



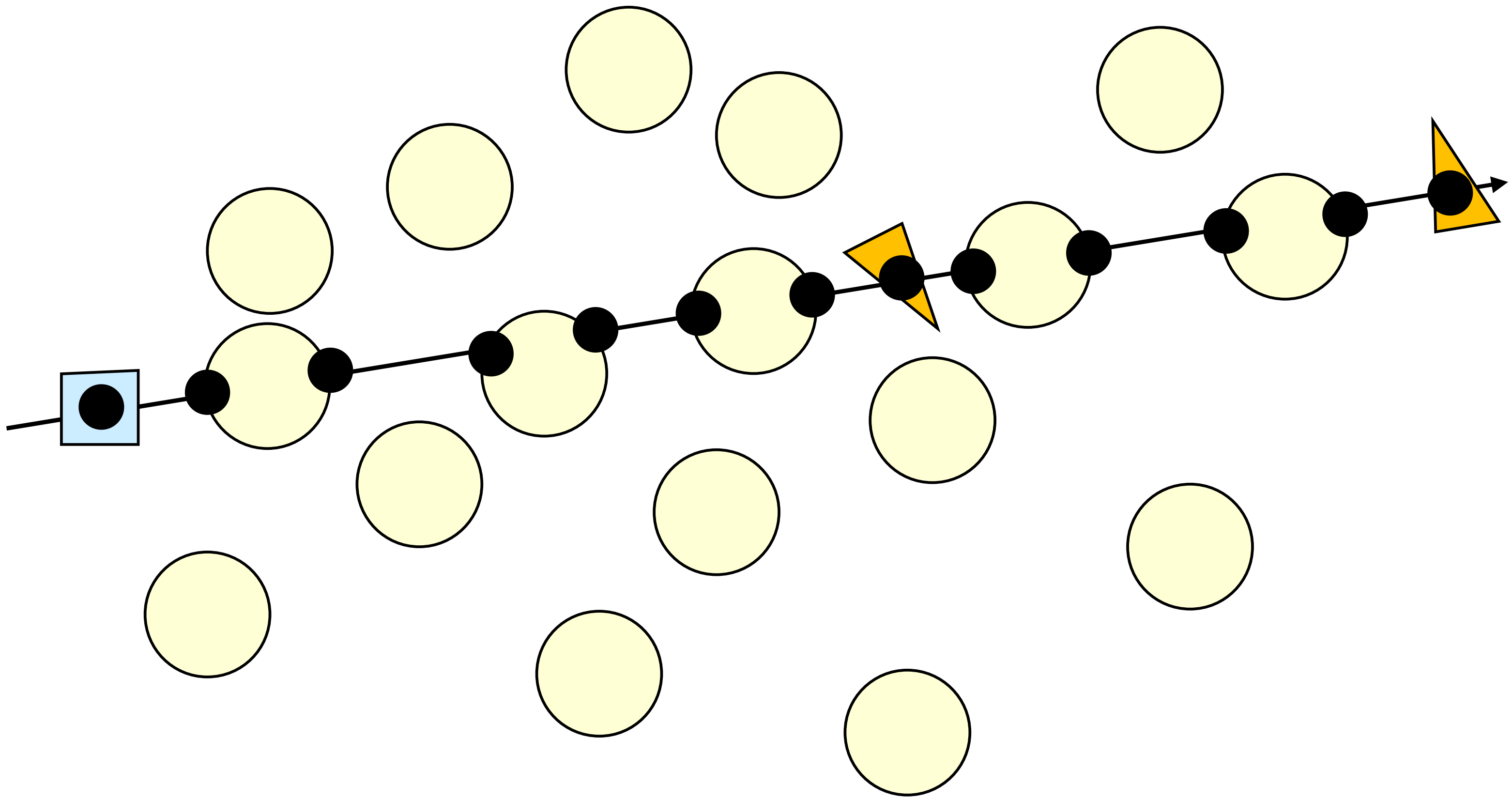
A cube is actually the intersection of 6 half-space planes (just 4 are shown here). Each of these will produce its own t intersection value. Treat them as pairs: (t_{x1},t_{x2}) , (t_{y1},t_{y2}) , (t_{z1},t_{z2})

The ultimate entry and exit values are:

$$\left.\begin{aligned} t_{min} &= \max(\ \min(t_{x1},\ t_{x2}),\ \min(t_{y1},\ t_{y2}),\ \min(t_{z1},\ t_{z2})\ ) \\ t_{max} &= \min(\ \max(t_{x1},\ t_{x2}),\ \max(t_{y1},\ t_{y2}),\ \max(t_{z1},\ t_{z2})\ ) \end{aligned}\right\}$$

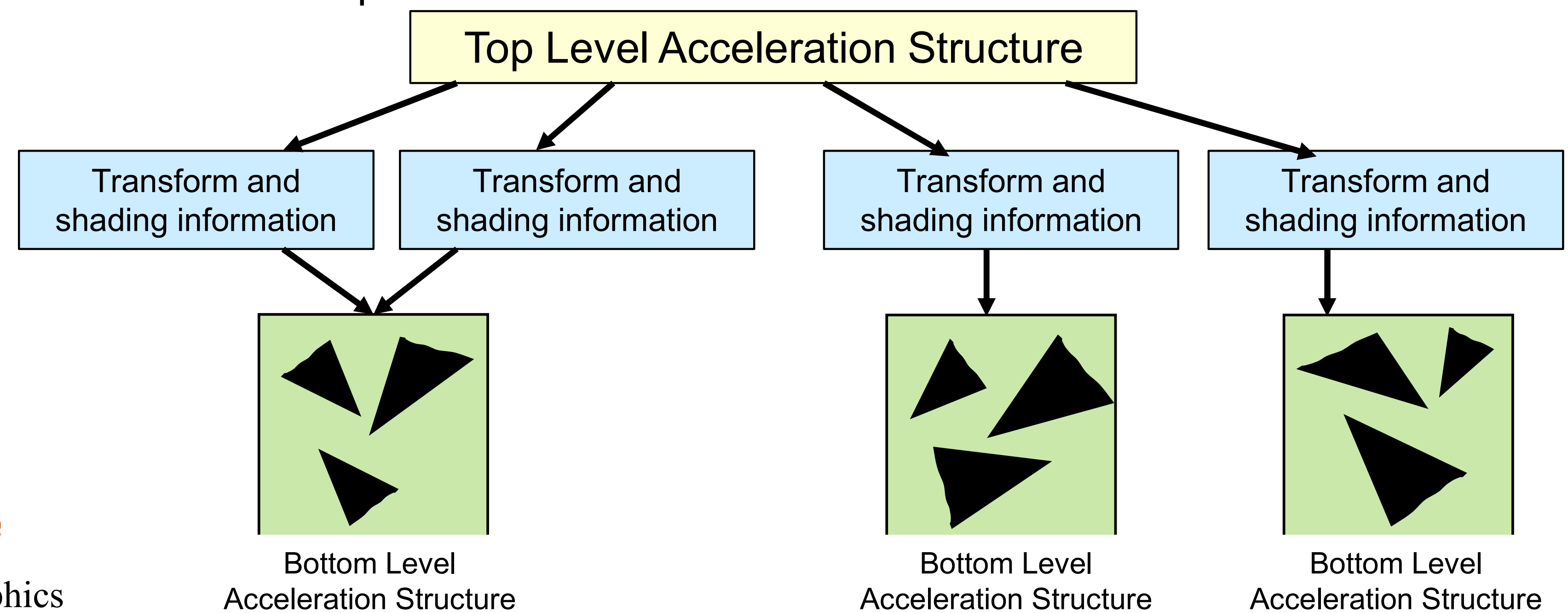
This works for all convex solids

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



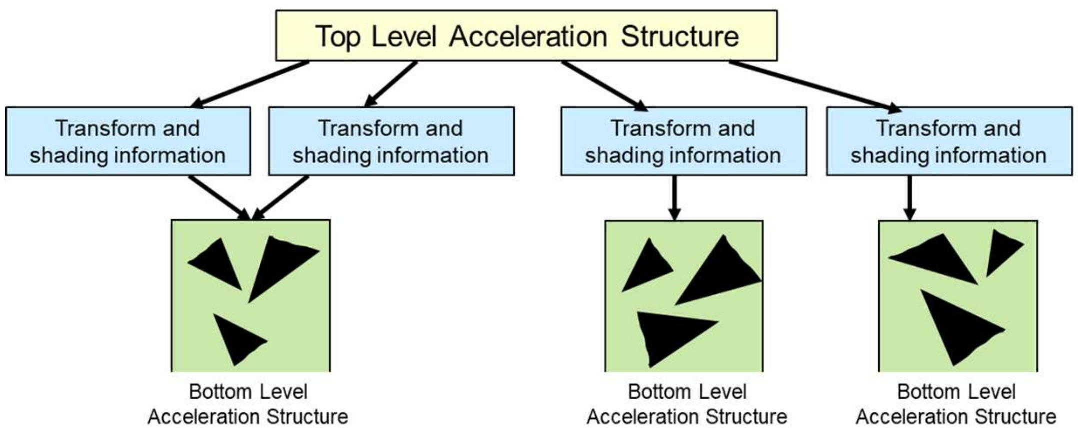
Acceleration Structures

- Bottom-level Acceleration Structure (BLAS) holds the vertex data and is built from vertex and index VkBuffers
- The BLAS can also hold transformations, but it looks like usually the BLAS holds vertices in the original Model Coordinates.
- Top-level Acceleration Structure (TLAS) holds a pointer to elements of the BLAS and a transformation.
- The BLAS is used as a Model Coordinate bounding box.
- The TLAS is used as a World Coordinate bounding box.
- A TLAS can instance multiple BLAS’s.



Creating *Bottom* Level Acceleration Structures

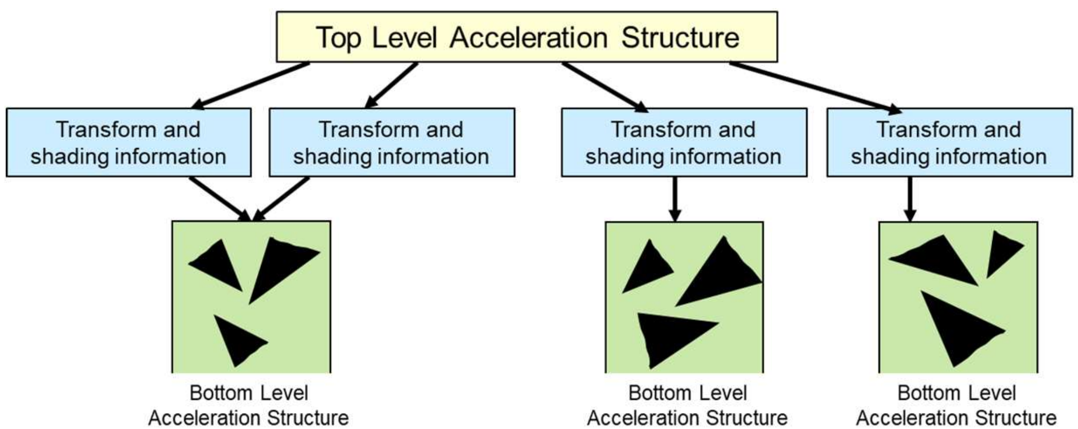

```
vkCreateAccelerationStructure                    BottomLevelAccelerationStructure;

VkAccelerationStructureInfo                      vasi;
        vasi.sType = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL;
        vasi.flags = 0;
        vasi.pNext = nullptr;
        vasi.instanceCount = 0;
        vasi.geometryCount = << number of vertex buffers >>
        vasi.pGeometries   = << vertex buffer pointers >>

 VkAccelerationStructureCreateInfo               vasci;
        vasci.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO;
        vasci.pNext = nullptr;
        vasci.info = &vasi;
        vasci.compactedSize = 0;

result = vkCreateAccelerationStructure( LogicalDevice, IN &vasci, PALLOCATOR, OUT &BottomLevelAcceleraionrStructure );
```



Oregon State University
Computer Graphics

Creating *Top* Level Acceleration Structures


```
vkCreateAccelerationStructure                     TopLevelAccelerationStructure;

VkAccelerationStructureInfo                       vasi;
        vasi.sType = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL;
        vasi.flags = 0;
        vasi.pNext = nullptr;
        vasi.instanceCount = << number of bottom level acceleration structure instances >>;
        vasi.geometryCount = 0;
        vasi.pGeometries   = VK_NULL_HANDLE;

 VkAccelerationStructureCreateInfo                vasci;
        vasci.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO;
        vasci.pNext = nullptr;
        vasci.info = &vasi;
        vasci.compactedSize = 0;

result = vkCreateAccelerationStructure( LogicalDevice, &vasci, PALLOCATOR, &TopLevelAcceleraionrStructure );
```



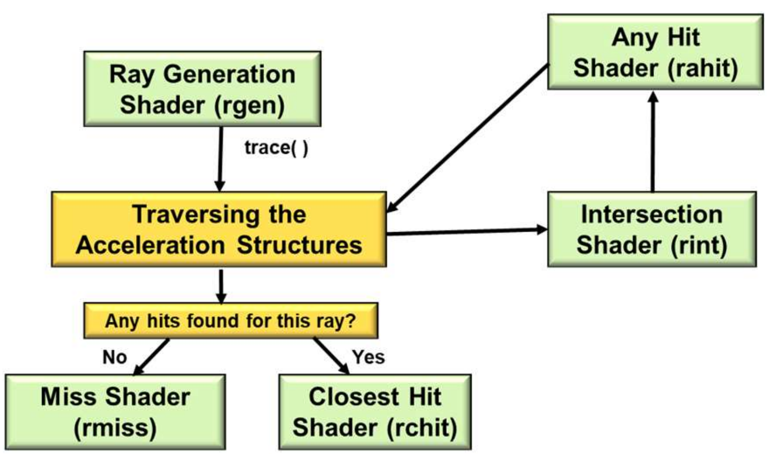
Oregon State University
Computer Graphics

Ray Generation Shader

Gets all of the rays going and writes the final color to the pixel

```
layout( location = 1 ) rayPayload  myPayLoad
{
        vec4 color;
};

void
main( )
{
        trace( topLevel, …, 1 );
        imageStore( framebuffer, gl_GlobalInvocationID.xy, color );
}
```

A "payload" is information that keeps getting passed through the process. Different stages can add to it. It is finally consumed at the very end, in this case by writing *color* into the pixel being worked on.



A New Built-in Function

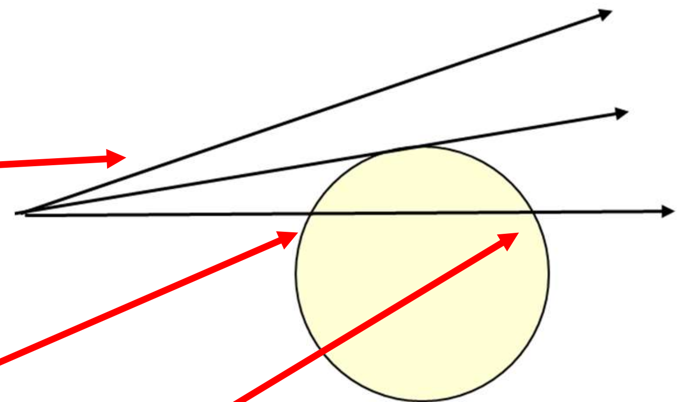
```
void trace
(
        accelerationStructure       topLevel,
        uint                        rayFlags,
        uint                        cullMask,
        uint                        sbtRecordOffset,
        uint                        sbtRecordStride,
        uint                        missIndex,
        vec3                        origin,
        float                       tmin,
        vec3                        direction,
        float                       tmax,
        int                         payload
);
```

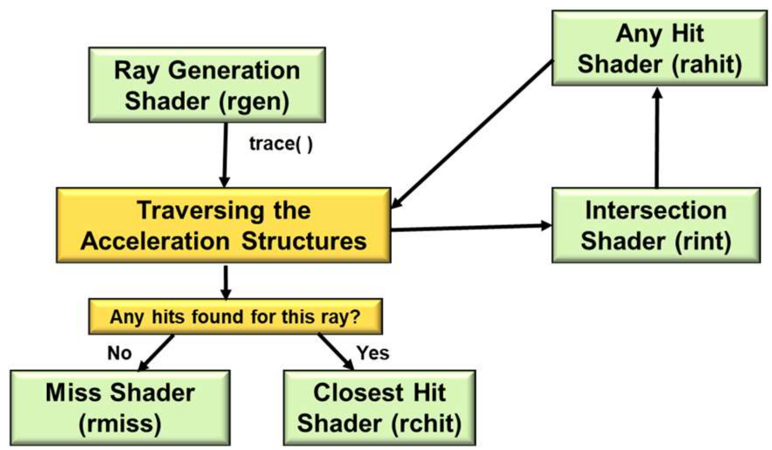
In Vulkan terms:
gl_WorldRayOrigin = (x_0,y_0,z_0)
gl_Hit = t
gl_WorldRayDirection = (dx,dy,dz)

Intersection Shader

Intersect a ray with an arbitrary 3D object.
Passes data to the Any Hit shader.
There is a built-in ray-triangle Intersection Shader.

```
hitAttribute  vec3  attribs

void main( )
{
        SpherePrimitive sph = spheres[ gl_PrimitiveID ];
        vec3 orig = gl_WorldRayOrigin;
        vec3 dir = normalize( gl_WorldRayDirection );
          . . .
        float discr = b*b – 4.*a*c;
        if( discr < 0. )
                return;

        float  tmp = ( -b - sqrt(discr) ) / (2.*a);
        if( gl_RayTmin < tmp  &&  tmp < gl_RayTmax )
        {
                vec3 p = orig + tmp * dir;
                attribs = p;
                reportIntersection( tmp, 0 );
                return;
        }
        tmp = ( -b + sqrt(discr) ) / (2.*a);
        if( gl_RayTmin < tmp  &&  tmp < gl_RayTmax )
        {
                vec3 p = orig + tmp * dir;
                attribs = p;
                reportIntersection( tmp, 0 );
                return;
        }
}
```

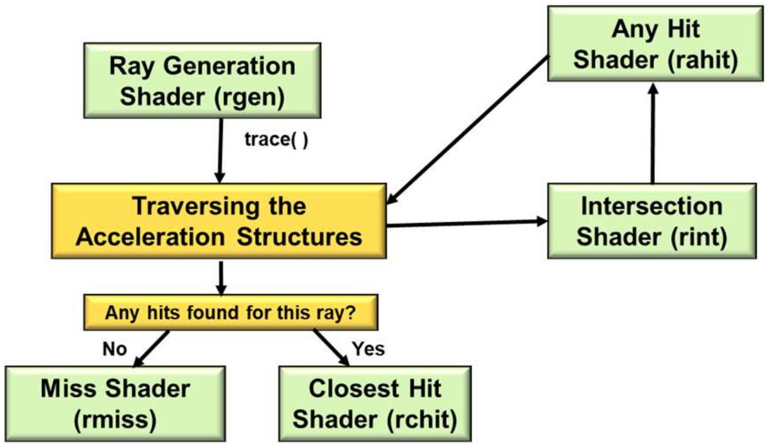




Miss Shader

Handle a ray that doesn't hit *any* objects

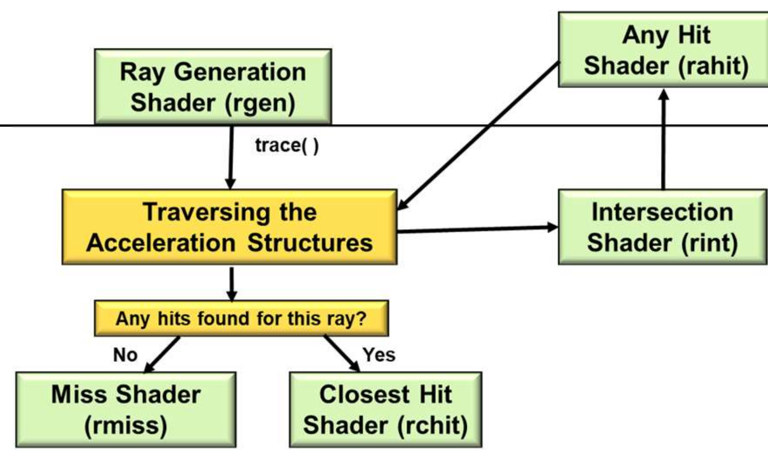
```
rayPayload myPayLoad
{
        vec4 color;
};

void
main( )
{
        color = vec4( 0., 0., 0., 1. );
}
```



Any Hit Shader

Handle a ray that hits *anything.*
Store information on each hit.
Can reject a hit.

```
layout( binding = 4, set = 0) buffer outputProperties
{
	float  outputValues[ ];
}  outputData;

layout(location = 0) rayPayloadIn uint outputId;
layout(location = 1) rayPayloadIn uint hitCounter;
hitAttribute  vec3 attribs;

void
main( )
{
	outputData.outputValues[ outputId + hitCounter ] = gl_PrimitiveID;
	hitCounter = hitCounter + 1;
}
```

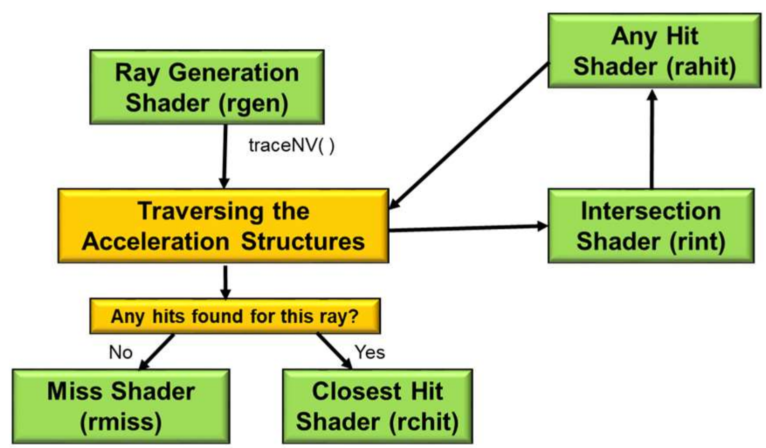


Closest Hit Shader

Handle the intersection closest to the viewer.
Collects data from the Any Hit shader.
Can spawn more rays.

```
rayPayload myPayLoad
{
        vec4 color;
};

void
main( )
{
        vec3 stp = gl_WorldRayOrigin + gl_Hit * gl_WorldRayDirection;
        color = texture( MaterialUnit, stp );        // material properties lookup
}
```

In Vulkan terms:
gl_WorldRayOrigin = (x_0,y_0,z_0)
gl_Hit = t
gl_WorldRayDirection = (dx,dy,dz)



Other New Built-in Functions

```
void terminateRay( );
```

```
void ignoreIntersection( );
```

Loosely equivalent to “discard”

```
void reportIntersection( float hit, uint hitKind );
```

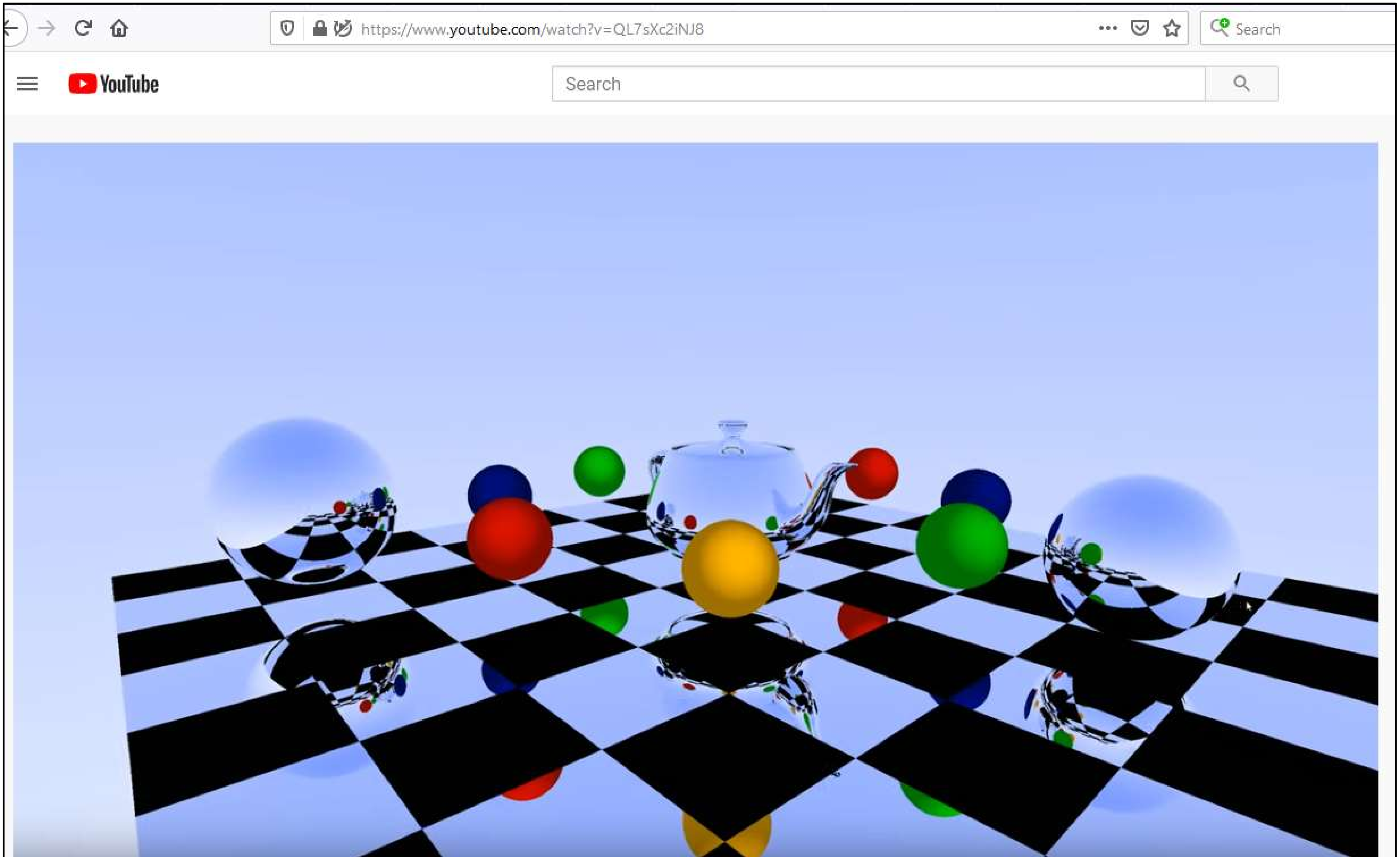
Ray Trace Pipeline Data Structure


```
VkPipeline                    RaytracePipeline;
VkPipelineLayout              PipelineLayout;

VkPipelineLayoutCreateInfo                vplci;
        vplci.sType                   = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
        vplci.pNext                   = nullptr;
        vplci.flags                   = 0;
        vplci.setLayoutCount          = 1;
        vplci.pSetLayouts             = &descriptorSetLayout;
        vplci.pushConstantRangeCount = 0;
        vplci.pPushConstantRanges     = nullptr;

result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, nullptr, OUT &PipelineLayout);


VkRayTracingPipelineCreateInfo            vrtpci;
        vrtpci.sType                  = VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO;
        vrtpci.pNext                  = nullptr;
        vrtpci.flags                  = 0;
        vrtpci.stageCount             = << # of shader stages in the ray-trace pipeline >>
        vrtpci.pStages                = << what those shader stages are >>
        vrtpci.groupCount             = << # of shader groups >>
        vrtpci.pGroups                = << pointer to the groups (a group is a combination of shader programs >>
        vrtpci.maxRecursionDepth  = << how many recursion layers deep the ray tracing is allowed to go >>;
        vrtpci.layout                 = PipelineLayout;
        vrtpci.basePipelineHandle     = VK_NULL_HANDLE;
        vrtpci.basePipelineIndex      = 0;

result = vkCreateRayTracingPipelines( LogicalDevice, PALLOCATOR, 1, IN &rvrtpci, nullptr, OUT &RaytracePipeline);
```

The Trigger comes from the Command Buffer: vlCmdBindPipeline( ) and vkCmdTraceRays( )

```
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_RAYTRACING, RaytracePipeline  );


vkCmdTraceRays(    CommandBuffer.
                   raygenShaderBindingTableBuffer, raygenShaderBindingOffset,
                   missShaderBindingTableBuffer,     missShaderBindingOffset,     missShaderBindingStride,
                   hitShaderBindingTableBuffer,      hitShaderBindingOffset,      hitShaderBindingStride,
                   callableShaderBindingTableBuffer, callableShaderBindingOffset, callableShaderBindingStride
                   width, height, depth );,
```

Check This Out!



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