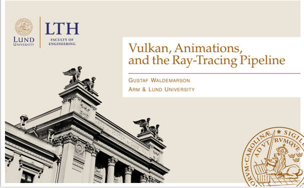
LUND UNIVERSITY

LTH
FACULTY OF ENGINEERING

Vulkan, Animations, and the Ray-Tracing Pipeline

Gustaf Waldemarson
Arm & Lund University

2023-12-05



LTH
Vulkan, Animations, and the Ray-Tracing Pipeline
Gustaf Waldemarson
Arm & Lund University

Hello everyone and welcome to this presentation. I believe Mike have already given you a brief introduction, but allow me to present myself. . .

Who am I?

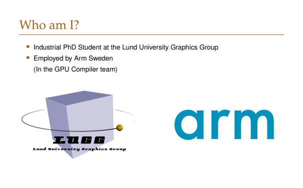
- Industrial PhD Student at the Lund University Graphics Group
- Employed by Arm Sweden

  (In the GPU Compiler team)

LUGG

Lund University Graphics Group

arm



Who am I?

- Industrial PhD Student at the Lund University Graphics Group
- Employed by Arm Sweden

  (In the GPU Compiler team)

My name is Gustaf Waldemarson, an industrial PhD employed by Arm. As such, my work is split between research here at Lund University and working for the Arm GPU compiler team up at Ideon.

As far as work goes, my research is primarily focused on various ray-tracing topics. Thus, today we figured we would give you all a very brief introduction into Vulkan and the newfangled ray tracing pipeline that you may have heard that many games have started to use.

That said, this presentation will mostly be an overview, so if you have any questions feel free to ask them as we go along.

Additionally, if there is time I will also talk a little about some more advanced animation techniques, such as how to interpolate and rotate cameras smoothly since some of my recent work have touched upon this.

Starting Off

What is Vulkan?

So lets start from the top shall we? What is Vulkan?

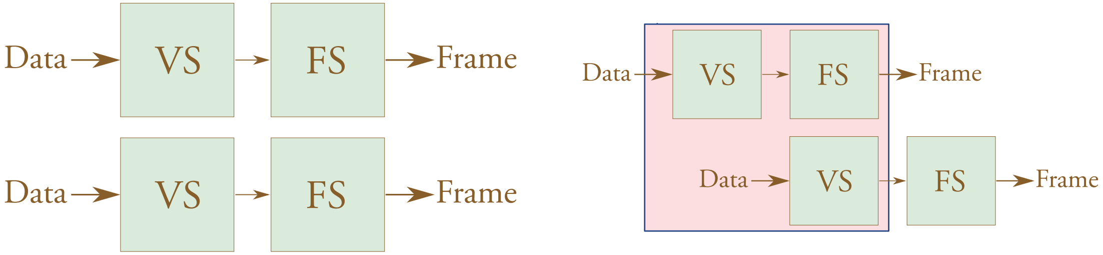
At its core, Vulkan is “just” another graphics API, just like OpenGL or DirectX. Which obviously begs the question, why bother? By now you all know OpenGL fairly well, so what is the point learning another framework to do essentially the same thing?

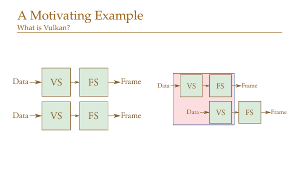
And the answer (at least historically), is performance. OpenGL and older versions of DX have many, many years of “API baggage” that they need to support since many applications depend on these features.

In broad terms then: Vulkan (and modern DX) lets the programmers get closer to the hardware, *potentially* unlocking more performance.

A Motivating Example

What is Vulkan?




A Motivating Example
What is Vulkan?

Just to give you all a motivating example. As you have seen up until now, GL application typically works on a frame-to-frame basis. This typically works just fine, but as you may already have seen in the earlier course, it is not uncommon to have an asymmetric load on the pipeline.

There is nothing in the GL application that tells the driver that “oh, and by the way you can go ahead and do some more vertex work”, so in order to achieve good performance, the driver must *guess* the intent of the code.

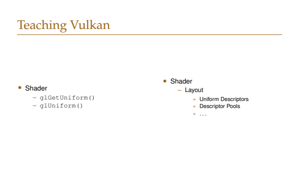
In Vulkan, these things are done explicitly from the get-go: It is up to the programmer to tell Vulkan exactly what dependencies are present such that it can efficiently pipeline the render jobs, such as in this facetious example.

I should add though that it *is* possible to write OpenGL code such that the effect will be the same. But to do that, the programmer must know what decisions the GPU driver will make, which of course, may be different from one GPU vendor to another. So if you have ever heard a game developer talk about a “driver fast-path”, this is what they are referring to.

Teaching Vulkan

- Shader
  - `glGetUniform()`
  - `glUniform()`

- Shader
  - Layout
    - Uniform Descriptors
    - Descriptor Pools
    - . . .



Teaching Vulkan

- Shader
  - glGetUniform()
  - glUniform()
- Shader
  - Layout
    - Uniform Descriptors
    - Descriptor Pools
    - . . .

This obviously begs the question: Why are we *not* teaching you graphics programming using Vulkan?

The main reason is simply time. As an example: In GL, to change the uniforms you need in a shader you update the shader, call `glGetUniform` and `glUniform` and you are pretty much done.

In `Vulkan`, such a change can be much more far-reaching: Changing the uniform also changes the “layout” of the shader, which may require you to allocate more descriptors, which may require allocating a larger descriptor pool and so on.

And, this is just to the equivalent of the `glGetUniform`, there is an equivalent amount of work needed to perform the equivalent action for the `glUniform` call.

This is all done to match more closely to what is actually done inside the driver and the hardware, and we could teach you these things, but they are pretty tedious, which means that much more work would be put on unnecessary details compared to actually teaching useful graphics concepts, such as deferred shading.

Starting Off

Vulkan Basis

Vulkan™

https://vulkan-tutorial.com/

G. Waldemarson

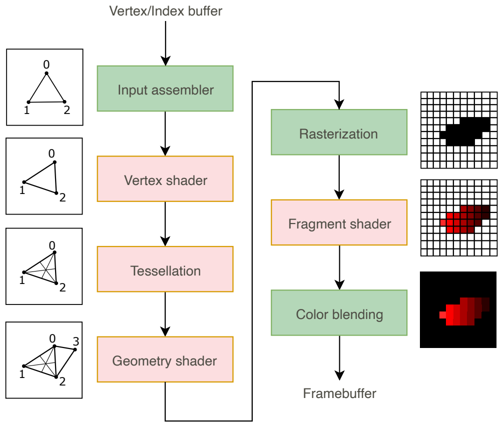
LUND UNIVERSITY

As I hope the previous example illustrated, describing the creation of a Vulkan app from scratch would take hours by itself, so I will not have time to go into any details.

That said, the most well known Vulkan tutorial is the one that you find along this (very unsurprisingly named) link, which walks through all the steps necessary to get a basic forward 3D renderer up and running. I went through this tutorial a few years ago and this served as an excellent starting point, and it is still being actively updated.

That said, you should know what you are getting into: No matter how you twist it, writing a Vulkan app from scratch is a long journey. E.g., you have to write almost 2000 lines of code before you even get a basic triangle on the screen. (A typical example for OpenGL is up and running in less than 100 lines). That said, you *will* learn a lot about how a GPU actually works by the end, so whether this is worth it is up to you to decide.

The Vulkan Pipelines



Vertex/Index buffer

Input assembler

Vertex shader

Tessellation

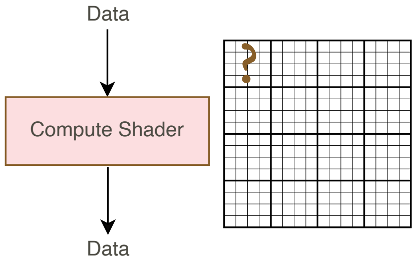
Geometry shader

Rasterization

Fragment shader

Color blending

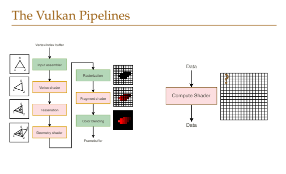
Framebuffer



Data

Compute Shader

Data



But now it is about time to change gears. Last week, Mike mentioned using OpenCL and other APIs to use the GPU for other, possibly non-graphics related tasks. In fact, almost all new graphics APIs have access to a similar kind of pipeline. Collectively, they are known as the compute-shader or compute-pipeline.

The typical use-case is pretty simple: Given some data in an array, operate on it and output data in another one.

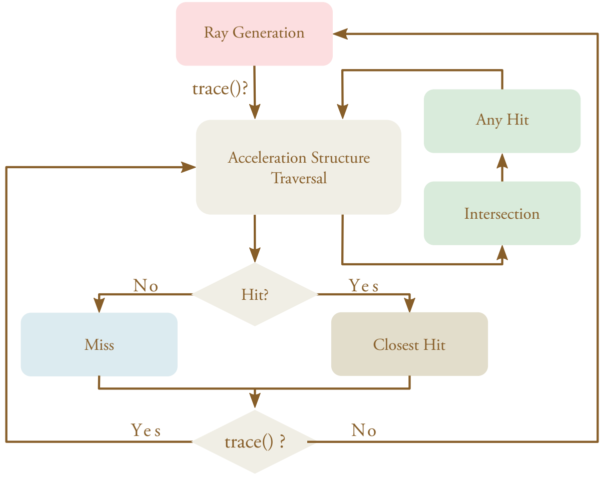
Compare this with the raster pipeline, which goes all the way from primitive vertices and indices through many stages before you finally arrive at some pixels in a framebuffer. In a compute-shader, the only thing you get from the pipeline is an index into some array with up to 3 dimensions, what you do with that is then entirely up to you to write in shader code.

As a concrete example, consider how a ray tracer is typically implemented: It walks from pixel to pixel sending out rays as it goes. This is often pretty easy to convert to a compute-shader, where the index represents the pixel it is currently sending rays from.

Naturally, `Vulkan` have access to this pipeline, and the compute-pipeline is in fact completely separated from the raster pipe, so the setup for it is actually a lot simpler.

In fact, the compute-pipeline is sometimes so much simpler to use that some games skip the raster pipeline entirely and *only* use compute-shaders for all rendering tasks.

The Ray Tracing Pipeline

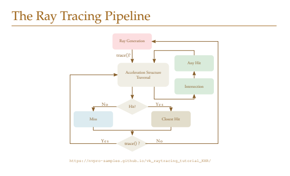


https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/

LUND UNIVERSITY

2023-12-05

└The Ray Tracing Pipeline



The raster- and compute-pipes are not the only ones, however. During the last few years, support has been added for a few more pipelines, one of which is naturally the ray tracing pipeline, so let us get into that one shall we?

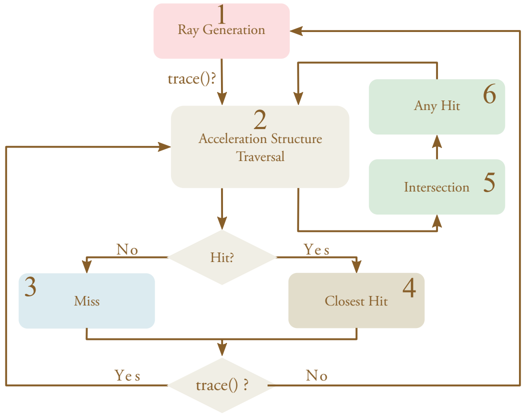
Now, instead of our instead of our raster pipeline from the previous slide, we will replace it entirely with this thing; which I hope you already guess is quite a bit more complicated than the other ones. In particular, I want to highlight *this*, up until now, the pipelines have always been fixed and linear, but now we are actually giving the GPUs the ability to loop inside the pipeline based on user input.

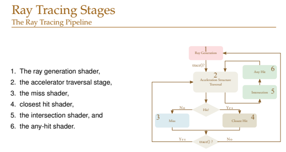
There is of course a bit of setup required to get this *Ray Tracing Pipeline* up and running such as building various data structures. But in the interest of time, I will skip over quite a few of these details. For the interested reader though, I can recommend the Nvidia tutorial on the subject, which I have linked down here, and I think we can make these slides and notes available later so that you can look things up if you are interested.

Ray Tracing Stages

The Ray Tracing Pipeline

1. The ray generation shader,
2. the accelerator traversal stage,
3. the miss shader,
4. closest hit shader,
5. the intersection shader, and
6. the any-hit shader.




Ray Tracing Stages
The Ray Tracing Pipeline
1. The ray generation shader,
2. the accelerator traversal stage,
3. the miss shader,
4. closest hit shader,
5. the intersection shader, and
6. the any-hit shader.

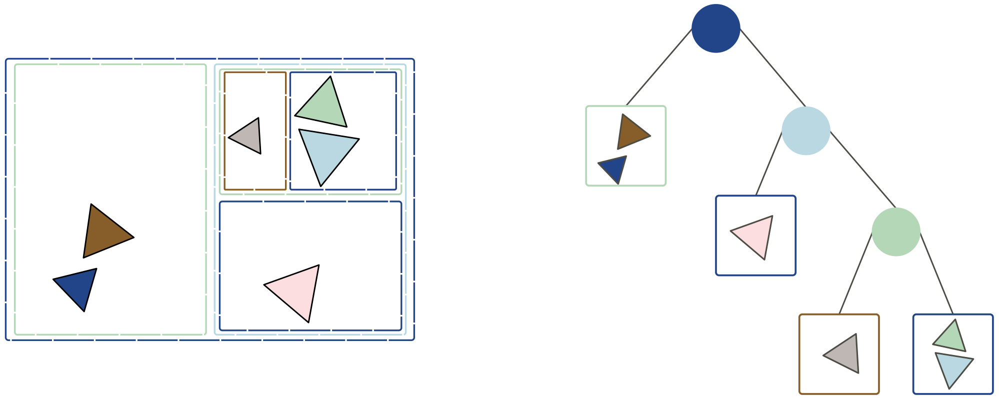
Essentially, in this new pipeline we have 6 new stages to contend with:

1. The ray generation shader,
2. the accelerator traversal stage,
3. the miss shader,
4. closest hit shader,
5. the intersection shader, and
6. the any-hit shader.

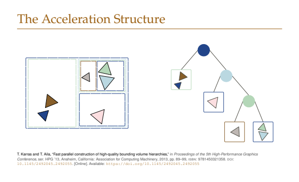
And we get to control all of these but the accelerator stage. Thankfully though, these shaders all follow the basic semantics used in compute shaders, so it is very straightforward to understand how to use each of these. In fact, writing the *shaders* is typically the easy part of using the ray tracing pipe.

I will not be able to do justice though. But I will go through each of them to give you an idea of the intent behind each stage.

The Acceleration Structure



T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG ’13, Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99, ISBN: 9781450321358. DOI: 10.1145/2492045.2492055. [Online]. Available: https://doi.org/10.1145/2492045.2492055



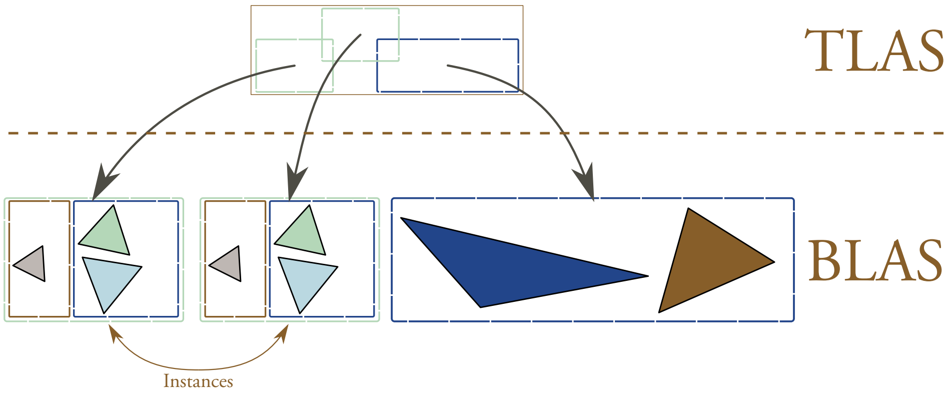
First though, we should probably talk about the *acceleration structure*. You have already gotten a taste of how important something like this is from the first lab, and historically, a lot of the efforts surrounding ray tracing has been focused around this structure.

However, for these modern APIs, it has been converted to more or less of a black box. What we get is what the API abstraction gives us: An opaque structure with two *levels*, each of which can be built or re-built separately, but rays can only be traced against the top level.

In practice, however, this structure is some kind of Bounding Volume Hierarchy, or BVH, that subdivides the search-space for each ray sent to it. The simplest analogue is a binary search tree, but for 3D space instead.

Note however where a normal binary tree typically have a unique shape for a given input, finding the “best” BVH for both 2D and 3D is believed to be an NP-hard problem [1], so a large amount of research has been poured into finding good heuristics to quickly build a decent one.

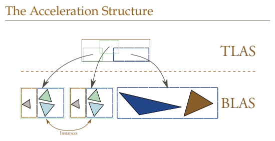
The Acceleration Structure



TLAS

BLAS

Instances



As I mentioned, these BVHs are split into two levels, appropriately named:

TLAS	Top-Level Acceleration Structure, and
BLAS	Bottom-Level Acceleration Structure

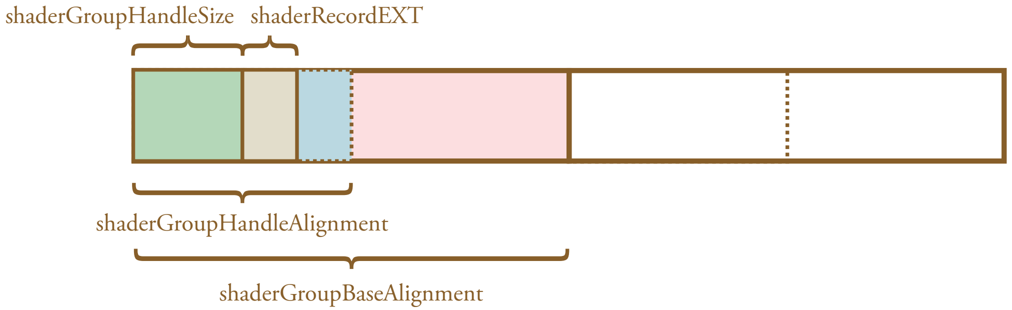
The creation of these structures is a bit complex since there are many low level details to consider, but at a high level it is pretty easy to see that we typically take our triangles and plop them into our BLASes, and then plop those BLASes into the TLAS. The typical linear algebra approach is also reused here: We use separate spaces for each object:

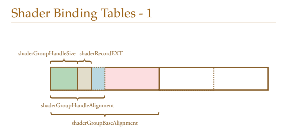
- TLAS-space typically represents world-space, whereas
- BLAS space represents Object-space, but this interpretation is up to the programmer.

Additionally, As you can see *here*, it is possible to re-use the same BLAS multiple times to perform instancing, which can greatly reduce the memory footprint for repeated objects.

Notice also the dependency: The TLAS cannot be built until all BLAS have been built. This often means that the TLAS is built using a different (faster) build algorithm.

Shader Binding Tables - 1




Shader Binding Tables - 1

There is also a somewhat non-obvious issue here: How do we know what shader to actually call?

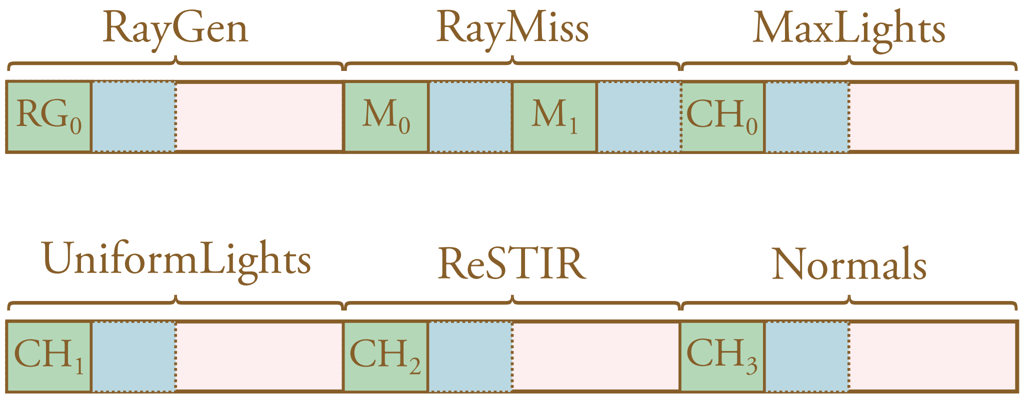
In the raster pipeline, the active shaders are always known (there is only one set of them after all), and those are "blindly" executed. If we want to use a different one for some objects, we just bind a new shader and use that instead.

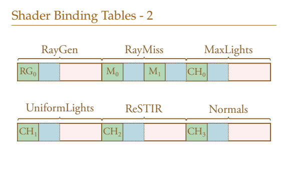
But in the ray tracing pipeline, we typically need all triangles at once to build the acceleration structure, and these may have different materials (or shaders) associated with them. So, we need to be able to call different shaders from *inside* the GPU. How do we handle something like that?

Solving this issue is the task of the so-called *shader binding table*: Essentially a jump table (or `vtable`) for shader calls during the ray tracing traversal. Interestingly though, it is entirely up to the programmer to construct this table, who must also adhere to several strict alignment requirements for the GPU to be able to use it, as seen in this example here.

(The programmer can even store some constant data for each shader in the `ShaderRecordEXT` segment.)

Shader Binding Tables - 2





As an example: In one of my applications I have structured my table like *this*. It has a single ray-gen shader, two miss-shaders, and four different closest hit-shaders that each do slightly different things.

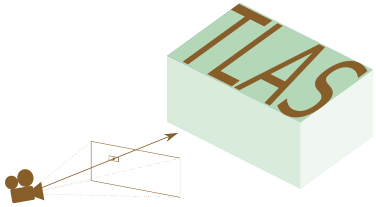
Thus, for this application the ray gen shader is only used once at the start to generate camera rays, but afterwards, the properties of the primitives or what the shaders themselves are doing can influence which shaders will be called next.

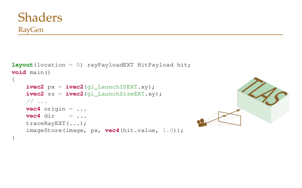
(Note that the hit-shaders are aligned to the *Base* alignment. This is required to allow this shader to be the first in an array.)

Shaders

RayGen

```
layout(location = 0) rayPayloadEXT HitPayload hit;
void main()
{
    ivec2 px = ivec2(gl_LaunchIDEXT.xy);
    ivec2 sz = ivec2(gl_LaunchSizeEXT.xy);
    // ...
    vec4 origin = ...
    vec4 dir    = ...
    traceRayEXT(...);
    imageStore(image, px, vec4(hit.value, 1.0));
}
```




Shaders
RayGen

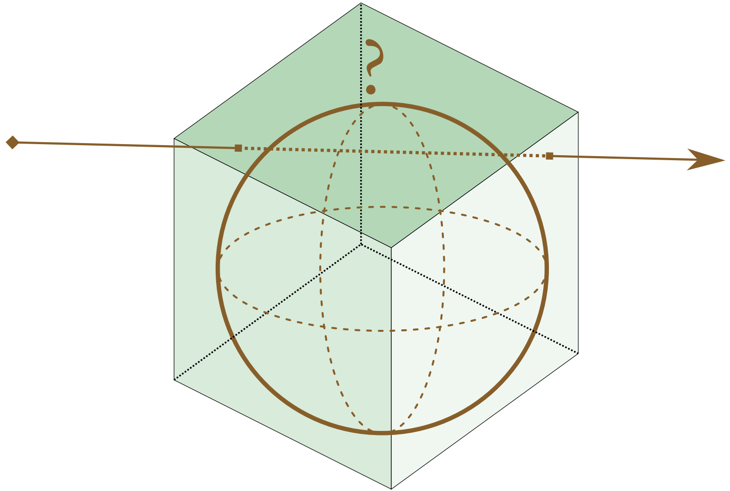
```
layout(location = 0) rayPayloadEXT HitPayload hit;
void main()
{
    ivec2 px = ivec2(gl_LaunchIDEXT.xy);
    ivec2 sz = ivec2(gl_LaunchSizeEXT.xy);
    // ...
    vec4 origin = ...
    vec4 dir    = ...
    traceRayEXT(...);
    imageStore(image, px, vec4(hit.value, 1.0));
}
```

Next I thought I would mention some details about each of the ray tracing shader stages:

First up is the ray generation shader, it is arguable the simplest of the bunch and the basic idea is easy: For each pixel, generate a camera ray and trace it through the Top-Level Acceleration Structure. Afterwards, the final color is indicated by the `hitPayload`, a small structure that is passed along the recursive calls.

This is just for my app however, in games this is often a bit more complicated, such as reading a GBuffer and only creating shadow rays, etc.

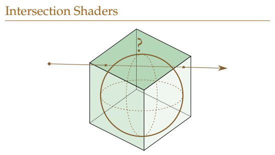
Intersection Shaders



G. Waldemarson

2023-12-05

Intersection Shaders

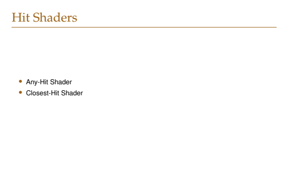

Intersection Shaders

Next in line is arguably the *intersection shader*. By default, Vulkan has a built-in implementation for intersecting triangles as this is the most common use case, but it is also possible to intersect *any* geometry of your choosing. And for that, we use Intersection shaders.

The main idea is that the acceleration structure is built to contain bounding boxes, then each of these boxes is associated with an intersection shader that is called during traversal: Given that we hit the bounding box, does the ray also hit the object inside it?

Hit Shaders

- Any-Hit Shader
- Closest-Hit Shader



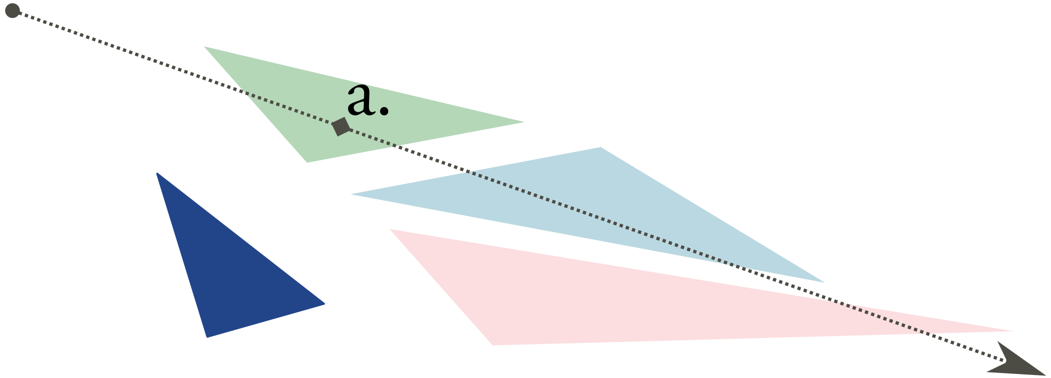
Hit Shaders

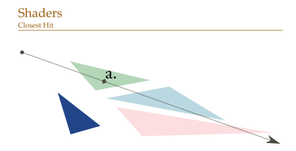
- Any-Hit Shader
- Closest-Hit Shader

Next we have the various types of “hit” shaders.

Shaders

Closest Hit

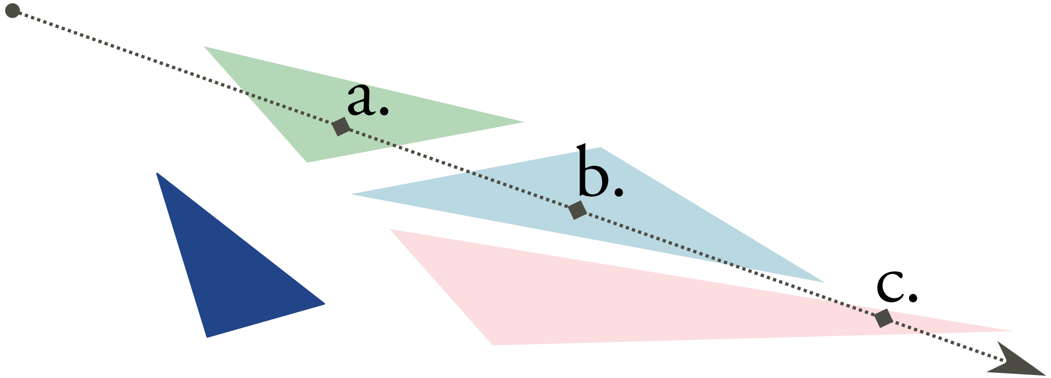


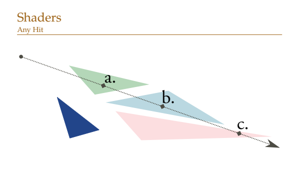


Starting with the closest hit: It is pretty unsurprisingly always going to return the closet hit point, such as `a` in this case simple case. However, *what* this hit-point (`a`) is up to the user to decided: He or she needs to set an appropriate `payload` value to represent it, as we will soon see.

Shaders

Any Hit





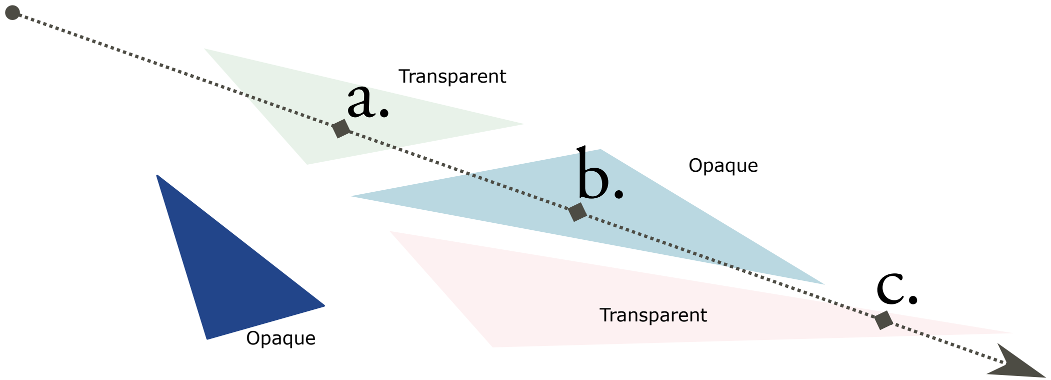
But as I mentioned before, it is not the only hit-shader; we also have the *any-hit* shader. This shader is primarily intended to implement some slightly more specialized types of operations, such as transparency; the way it works is that given a ray going through some hierarchy, the shader can be called for *any* hit point along the path of the ray.

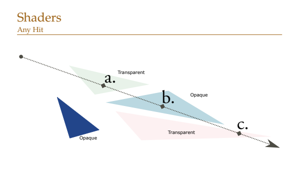
The primary task for this shader is then to perform some computations, then decide whether to *commit* the intersection or not. Once committed, we effectively change the ray `t_max`, meaning that we cannot hit anything after it.

Note though that it is not defined *which* hit is called first, it could be *any* hit along the path (such as *a* in this case) but the implementation can return whichever of *a*, *b* or *c*, and once a hit is ‘committed’, no hit ‘behind’ it is reachable.

Shaders

Any Hit



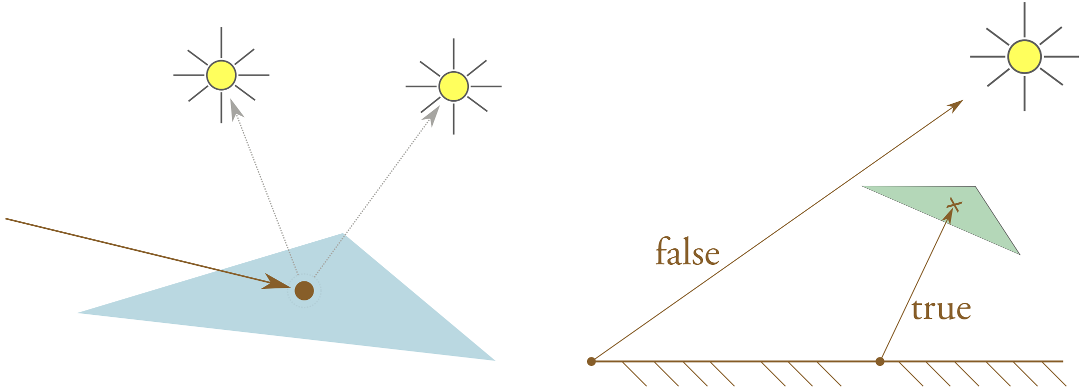

Shaders
Any Hit
a. Transparent
b. Opaque
c. Transparent
Opaque

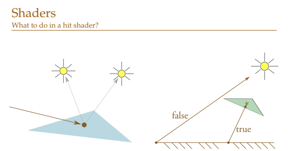
This is typically used to implement some kind of transparency where we do not care about refraction, but there are all manner of things that can be done.

As an example, consider the case where *b* is opaque, but *a* and *c* are partially transparent. Further, let's say that the traversal first calls the *any-hit* shader on *c*. Thus, we see that we have a color that we should *probably* blend into our final color. Next, let's say that we hit *b*. Since this is opaque, we no longer care about anything behind it, so the previous result can be discarded. Finally, we hit *a*, which gives us a color that we should blend with the opaque one.

Shaders

What to do in a hit shader?




Shaders
What to do in a hit shader?

But what do we actually do in a hit-shader? Well, that is really up to you, but as a simple example you could apply a normal phong shading by looking up the direction to some lights, or you could send a new ray recursively towards some light source to get a realistic shadowing effect.

This also opens up somewhat non-obvious issue: What exactly is a “hit”?

Unfortunately for us, there is really no free-lunch, we only get the bare minimum of information:

For triangle shaders, we only get two things: The ray `t` value of the hit and the barycentric coordinates. Any other data must be computed or fetched separately. So if we need the normals to apply our shading, we need to fetch those and perform barycentric interpolation ourselves!

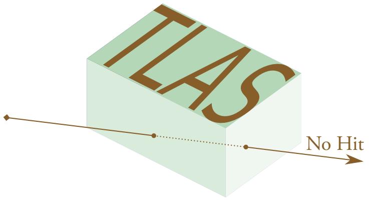
(For intersection shaders, it is up to the programmer to specify what values will be returned.)

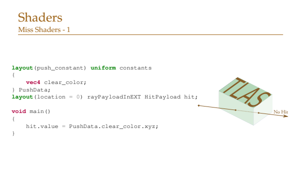
Shaders

Miss Shaders - 1

```
layout(push_constant) uniform constants
{
    vec4 clear_color;
} PushData;
layout(location = 0) rayPayloadInEXT HitPayload hit;

void main()
{
    hit.value = PushData.clear_color.xyz;
}
```




Shaders
Miss Shaders - 1

layout(push_constant) uniform constants
{
    vec4 clear_color;
} PushData;
layout(location = 0) rayPayloadInEXT HitPayload hit;

void main()
{
    hit.value = PushData.clear_color.xyz;
}

TLAS
No Hit

So, if we have hit-shaders, *obviously* we have to have the opposite: *miss* shaders. The idea is essentially if we do not actually hit something, this shader will be called instead.

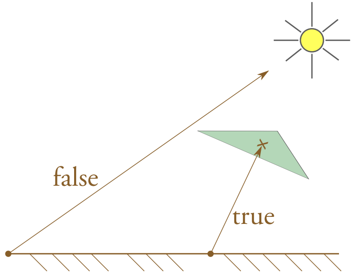
One examples of this in use is when we want to apply a fixed color when we don't hit anything in the scene, such as this shader here.

Shaders

Ray Miss - 2

```
layout(location = 1) rayPayloadInEXT bool in_shadow;

void main()
{
    in_shadow = false;
}
```



Shaders
Ray Miss - 2

layout(location = 1) rayPayloadInEXT bool in_shadow;
void main()
{
    in_shadow = false;
}

Another is when we want to add shadows to the object:

From the closest hit, we recursively trace a ray towards a light source, if that ray *misses*, the light sources is actually in full view from the hit point, so it is *not* in shadow. Thus, we have a miss shader that simply informs the closest hit (where we recursed from) that we are *not* in shadow.

And that is effectively the new shaders. And these all need to work together from the shader binding table to properly *run* the hardware ray-tracing pipeline.

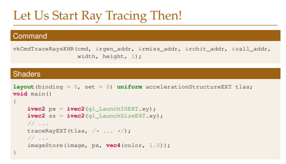
Let Us Start Ray Tracing Then!

Command

```
vkCmdTraceRaysKHR(cmd, &rgen_addr, &rmiss_addr, &rchit_addr, &call_addr,
                  width, height, 1);
```

Shaders

```
layout(binding = 0, set = 0) uniform accelerationStructureEXT tlas;
void main()
{
    ivec2 px = ivec2(gl_LaunchIDEXT.xy);
    ivec2 sz = ivec2(gl_LaunchSizeEXT.xy);
    // ...
    traceRayEXT(tlas, /* ... */);
    // ...
    imageStore(image, px, vec4(color, 1.0));
}
```



Let Us Start Ray Tracing Then!

Command

vkCmdTraceRaysKHR(cmd, &rgen_addr, &rmiss_addr, &rchit_addr, &call_addr, width, height, 1);

Shaders

layout(binding = 0, set = 0) uniform accelerationStructureEXT tlas; void main() { ivec2 px = ivec2(gl_LaunchIDEXT.xy); ivec2 sz = ivec2(gl_LaunchSizeEXT.xy); // ... traceRayEXT(tlas, /* ... */); // ... imageStore(image, px, vec4(color, 1.0)); }

So, to actually start ray tracing we do this:

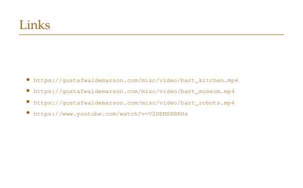
In Vulkan call, `vkCmdTraceRaysKHR`, and give it the *starting* address for all shader types, as well as how many rays to send (in a width/height/depth fashion, same as for compute shaders).

Next, the RayGen shader will start executing, which will of course, generate rays for the ray tracing process; which is done by calling the `TraceRayEXT` function in the GLSL code. This call receives which acceleration structure to use, what ray to use as well as some flags, which I have omitted here for brevity.

Finally, when everything has done its things with recursions and so forth, we will return to this shader, where we usually have the final color that we want to write out somewhere, typically to a texture or frame-buffer.

Links

- https://gustafwaldemarson.com/misc/video/bart_kitchen.mp4
- https://gustafwaldemarson.com/misc/video/bart_museum.mp4
- https://gustafwaldemarson.com/misc/video/bart_robots.mp4
- https://www.youtube.com/watch?v=V2DEMSRBKHs



Links

- https://gustafwaldemarson.com/misc/video/bart_kitchen.mp4
- https://gustafwaldemarson.com/misc/video/bart_museum.mp4
- https://gustafwaldemarson.com/misc/video/bart_robots.mp4
- https://www.youtube.com/watch?v=V2DEMSRBKHs

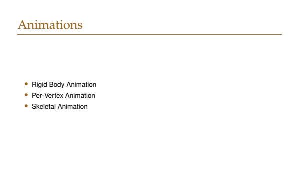
Running live-demos rarely works, and in this case, I don't even have a machine that can run the demo, (only new Macs actually get ray-tracing). Thus, let us see if a recording can be used videos can be found here. These are of course my personal, and arguably simplistic applications.

If we instead look at a much better demo from some who have had a bit more man-hours available, we can get some pretty impressive ray tracing action.

Nvidia says that this demo is entirely path-traced in real time at 4k-60FPS, which is quite impressive to say the least.

Animations

- Rigid Body Animation
- Per-Vertex Animation
- Skeletal Animation



Animations

- Rigid Body Animation
- Per-Vertex Animation
- Skeletal Animation

But that is all the details about the ray-tracing pipeline for now. Next, I would like to talk a bit about animations.

Typically, I work with offline ray-tracing as such animation is not as big of a concern, but when it starts moving over to real-time, one need to start to consider how to handle various types of movement as well.

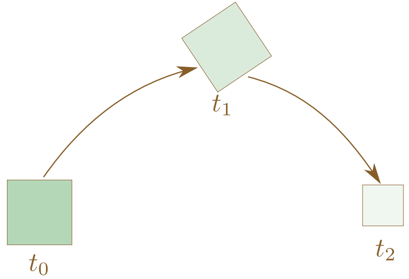
Naturally though, there several types of animations all with different pros and cons. The most common is undoubtedly the so-called *Rigid-Body-Animation*, which is what you have worked with the most in these two courses.

Beyond that, we have at least two major kinds of animations: Per-Vertex- and Skeletal Animations.

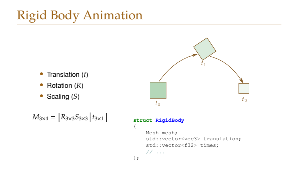
Rigid Body Animation

- Translation (t)
- Rotation (R)
- Scaling (S)

$$M_{3\times4} = \left[R_{3\times3}S_{3\times3} \,\middle|\, t_{3\times1}\right]$$



```
struct RigidBody
{
    Mesh mesh;
    std::vector<vec3> translation;
    std::vector<f32> times;
    // ...
};
```


Rigid Body Animation

- Translation (t)
- Rotation (R)
- Scaling (S)

$M_{3\times4} = \left[R_{3\times3}S_{3\times3} \,\middle|\, t_{3\times1}\right]$

struct RigidBody
{
Mesh mesh;
std::vector<vec3> translation;
std::vector<f32> times;
// ...
};

But for completeness, let us start with rigid bodies: I think this may be familiar to most, but just in case:

There are 3 things that encompasses a rigid body transformation:

- Translations in space,
- Rotations,
- and Scaling, or Skewing operations.

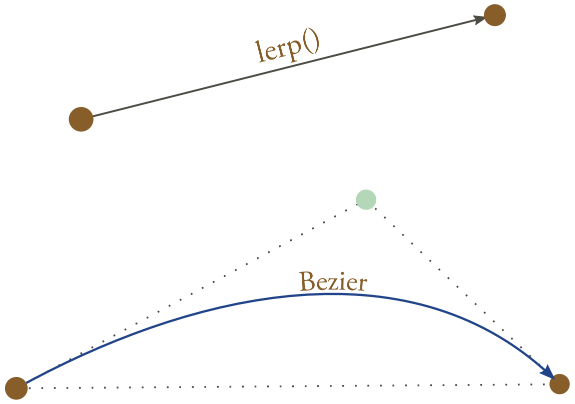
Thus, any rigid body transform can thus be represented using a 4×3 matrix.

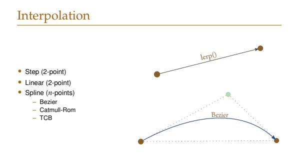
However, if you want to animate the transformations, these quantities must be kept separate when interpolating and then combined when doing the actual rendering.

(It may be of interest to know that each of the transform 'states' are typically referred to as "keyframes", or simply "keys".)

Interpolation

- Step (2-point)
- Linear (2-point)
- Spline (n-points)
  - Bezier
  - Catmull-Rom
  - TCB




Interpolation

- Step (2-point)
- Linear (2-point)
- Spline (n-points)
  - Bezier
  - Catmull-Rom
  - TCB

lerp()
Bezier

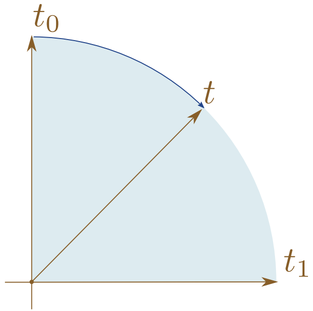
For completeness, I've included this slide about interpolations. That is, an algorithm for computing the values between two other values. And as this list hints at, there are *many* interpolation algorithms out there. Linear interpolation is well known, but for smooth animation variations of Bezier curves are commonly.

Suffice to say that my demo uses the so called TCB splines, since that is what the scenes I'm using have data for. The details of this type of spline are not really important for this presentation, however.

Interpolating Rotations

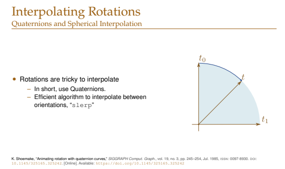
Quaternions and Spherical Interpolation

- Rotations are tricky to interpolate
  - In short, use Quaternions.
  - Efficient algorithm to interpolate between orientations, “`slerp`”



K. Shoemake, “Animating rotation with quaternion curves,” *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 245–254, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325242. [Online]. Available: https://doi.org/10.1145/325165.325242

└Interpolating Rotations


Interpolating Rotations
Quaternions and Spherical Interpolation

- Rotations are tricky to interpolate
  - In short, use Quaternions.
  - Efficient algorithm to interpolate between orientations, “slerp”

K. Shoemake, “Animating rotation with quaternion curves,” SIGGRAPH Comput. Graph., vol. 19, no. 3, pp. 245–254, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325242. [Online]. Available: https://doi.org/10.1145/325165.325242

Interpolating between locations and scaling factors is pretty easy, but it may not be obvious is how to interpolate *rotations*.

In short, one uses the Quaternion based algorithm known as `slerp` or “spherical interpolation”.

As a descriptive example: Consider the task of interpolating between two points on a circle. You cannot linearly interpolate, as that would “cross” the circle. Instead, you have to use some trigonometry to correctly find the point along the arc. This is essentially what the `slerp` algorithm does.

Interpolating Rotations

Quaternions and Spherical Interpolation

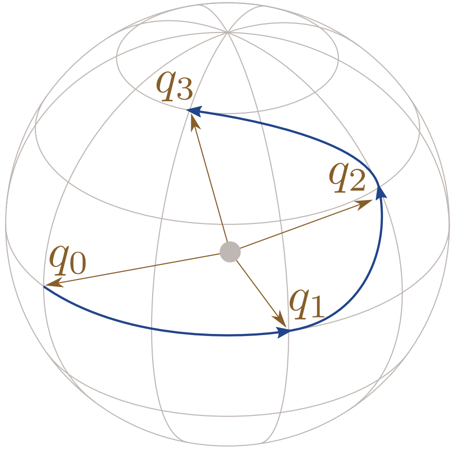


Figure: 3D spherical interpolation example

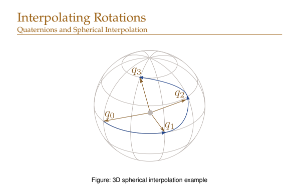
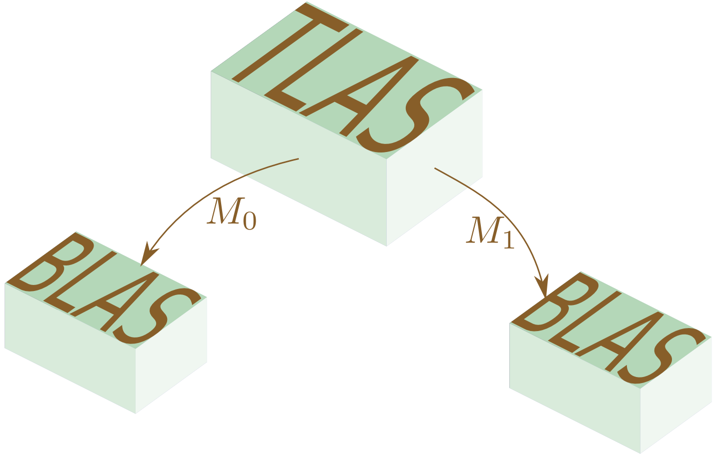
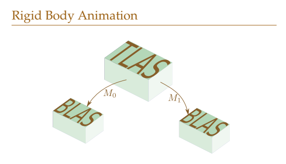

Interpolating Rotations
Quaternions and Spherical Interpolation

Figure: 3D spherical interpolation example

Typically though, we are interested in the 3D rotations, which is the primary reason to use Quaternions. Without them, computing this kind of orientation change would be pretty tricky.

Rigid Body Animation



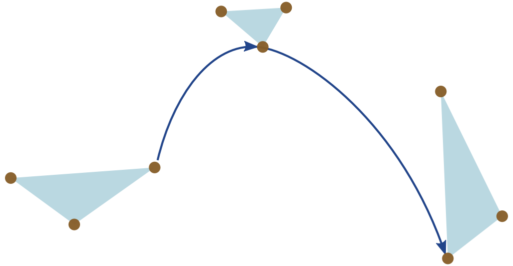

Rigid Body Animation

As far as Vulkan is concerned with however, rigid body transforms fits directly into the ray tracing API. When constructing the acceleration structure, each BLAS is associated with a rigid-body transform during construction, thus animating this type of transform is trivial and only requires rebuilding of the TLAS each frame.

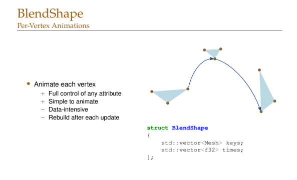
BlendShape

Per-Vertex Animations

- Animate each vertex
  - + Full control of any attribute
  - + Simple to animate
  - – Data-intensive
  - – Rebuild after each update



```
struct BlendShape
{
    std::vector<Mesh> keys;
    std::vector<f32> times;
};
```



BlendShape
Per-Vertex Animations

- Animate each vertex
  - + Full control of any attribute
  - + Simple to animate
  - – Data-intensive
  - – Rebuild after each update

```
struct BlendShape
{
    std::vector<Mesh> keys;
    std::vector<f32> times;
};
```

Next we have the so-called Per-Vertex Animations, which often go by the name of `BlendShapes` or `MorphTargets` in games.

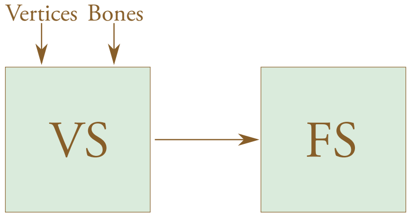
These are rather simple to describe: Each attribute is allowed to be arbitrarily deformed, so we just interpolate between them, effectively giving us full control of the animation with a relatively easy algorithm (implementation wise). However, as you can imagine, per-vertex animations requires duplicating each mesh for each of the keys and must also operate on *all* attributes on the selected keys, which may be very data intensive for large meshes.

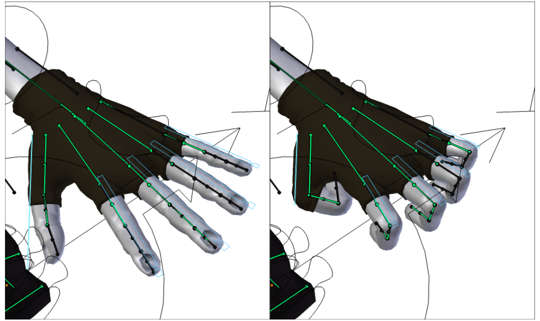
Additionally, since the animation is entirely arbitrary, the BLAS that contains it must also be rebuilt after each animation step.

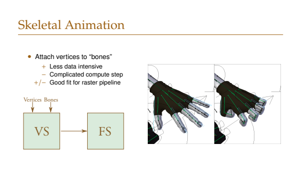
But, it is still the favored approach for some complex animations with a lot of motion, such as accurate facial animations.

Skeletal Animation

- Attach vertices to “bones”
  - + Less data intensive
  - – Complicated compute step
  - +/– Good fit for raster pipeline







Skeletal Animation

- Attach vertices to “bones”
  - + Less data intensive
  - – Complicated compute step
  - +/– Good fit for raster pipeline

Lastly, we have skeletal animations, (also known as *skinning*), as the name suggests, it is based around the animation of “bones” that the vertices are attached to.

In short: This makes for a less data-intensive animation process, but makes the animation algorithm a bit more complicated. Also, creating the animation itself, a process typically referred to as *rigging* is also quite a bit more complicated.

Traditionally though, this has been used very effectively on the raster pipeline, since this animation typically maps pretty well to the vertex shader. Unfortunately, there is no such mapping between the current ray tracing APIs and skeletal animations, which leaves us with essentially the same drawbacks as for the `BlendShapes`.

Thanks for Listening

Questions

- Thanks for listening!
- Questions and Answers

?

Thanks for Listening
Questions

- Thanks for listening!
- Questions and Answers

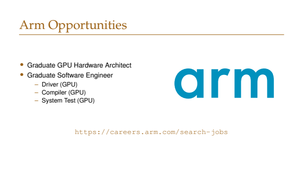
?

But that is about all I have time for, so thanks for listening; if you have any other question, Vulkan, animation, ray-tracing or otherwise, feel free to ask!

Arm Opportunities

- Graduate GPU Hardware Architect
- Graduate Software Engineer
  - Driver (GPU)
  - Compiler (GPU)
  - System Test (GPU)

https://careers.arm.com/search-jobs



Arm Opportunities

- Graduate GPU Hardware Architect
- Graduate Software Engineer
  - Driver (GPU)
  - Compiler (GPU)
  - System Test (GPU)

https://careers.arm.com/search-jobs

The last thing I want to mention is that Arm here in Lund is also hiring, with a decent number of graduate and intern positions available. In particular, the guys that create the *virtual* model of the GPU, whom for some reason are called GPU *architects* over at Arm have a number of open positions, as do the GPU driver, compiler and system testers.

But there are also quite a few other positions available, including interning opportunities. I'd recommend just following the link here and searching for graduate or intern positions in Lund.

Ray Tracing Links

- https://www.gsn-lib.org/docs/nodes/raytracing.php

Ray Tracing Links

- https://www.gsn-lib.org/docs/nodes/raytracing.php

The End

2023-12-05

References

[1] T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG ’13, Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99, ISBN: 9781450321358. DOI: 10.1145/2492045.2492055. [Online]. Available: https://doi.org/10.1145/2492045.2492055.

[2] K. Shoemake, “Animating rotation with quaternion curves,” *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 245–254, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325242. [Online]. Available: https://doi.org/10.1145/325165.325242.

References

[1] T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG ’13, Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99, ISBN: 9781450321358. DOI: 10.1145/2492045.2492055. [Online]. Available: https://doi.org/10.1145/2492045.2492055.

[2] K. Shoemake, “Animating rotation with quaternion curves,” *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 245–254, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325242. [Online]. Available: https://doi.org/10.1145/325165.325242.

[1] T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG ’13, Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99, ISBN: 9781450321358. DOI: 10.1145/2492045.2492055. [Online]. Available: https://doi.org/10.1145/2492045.2492055.

[2] K. Shoemake, “Animating rotation with quaternion curves,” *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 245–254, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325242. [Online]. Available: https://doi.org/10.1145/325165.325242.

Leftover Slides

G. Waldemarson

LUND UNIVERSITY

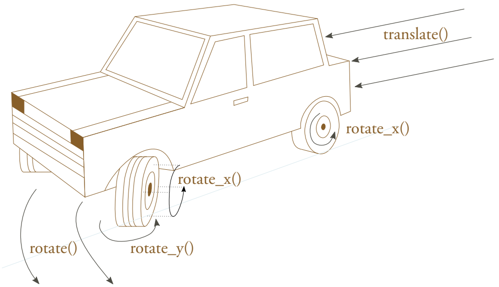
2023-12-05

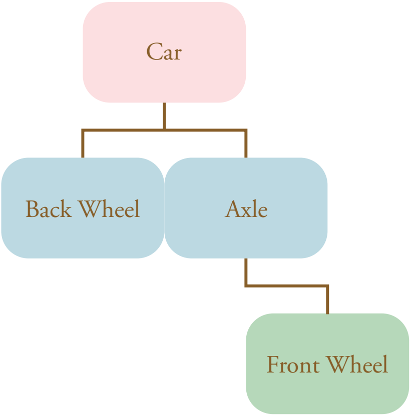
└Leftover Slides

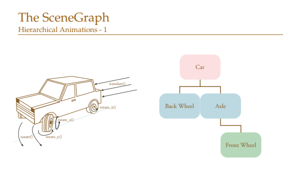
Leftover Slides

The SceneGraph

Hierarchical Animations - 1





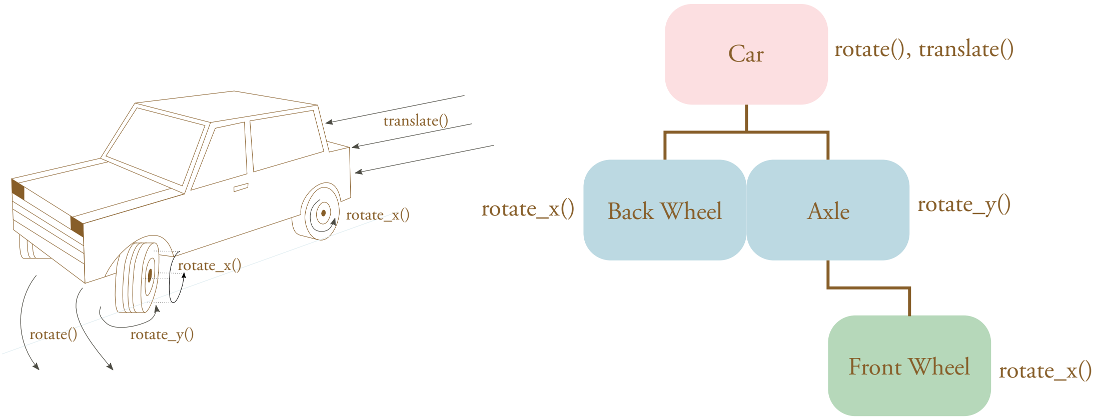


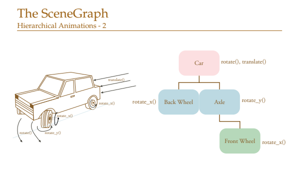
Animating single objects is relatively easy, however, objects are often connected into a hierarchy, and when we move some high-level object, we want all objects connected to that one to follow along.

This is all captured in the datastructure commonly referred to as the `SceneGraph`.

The SceneGraph

Hierarchical Animations - 2



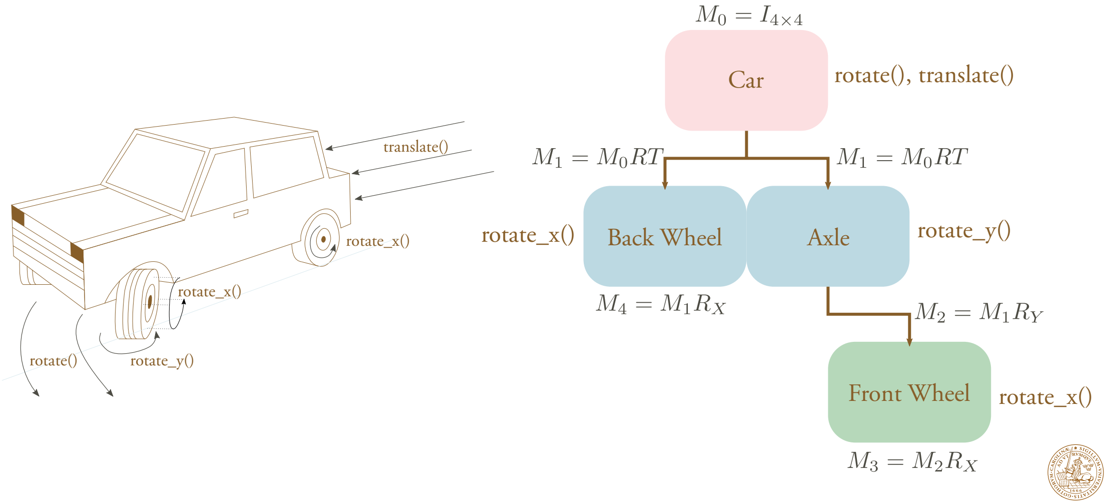


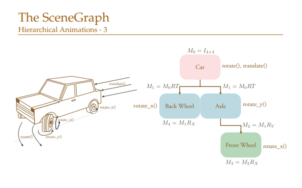
Consider this example: We want to simulate a car that turns left. To do that, we need to do the following:

1. Translate and rotate the car chassi.
2. Rotate the front axle the front wheel is connected to (Probably the Y axis).
3. Rotate the wheels (probably around the X axis).

The SceneGraph

Hierarchical Animations - 3





Eventually though, we need to convert all of this to something that we can send to the GPU. Thus, we start with an identity transform and simply traverse the Scenegraph in depth-first order, applying each transform in the same order, until each node has been assigned a transform.

(The astute among you may even notice some parallelism: We can walk from the leaves upwards instead if we allow some redundant computations.)