Sample Code

Encoding argument buffers on the GPU

Use a compute pass to encode an argument buffer and access its arguments in a subsequent render pass.

Download

iOS 14.0+ | iPadOS 14.0+ | macOS 10.13+ | Xcode 14.0+

Overview

In Using argument buffers with resource heaps, you learned how to combine argument buffers with arrays of resources and resource heaps.

In this sample, you'll learn how to encode resources into argument buffers with a graphics or compute function. In particular, you'll learn how to write data into an argument buffer from a compute pass and then read that data in a render pass. The sample renders a grid of multiple quad instances with two textures applied to each, where the textures slide from left to right within the quad and move from left to right between quads.

Getting started

The sample can run only on devices that support Tier 2 argument buffers. Tier 2 devices allow graphics or compute functions to encode data into an argument buffer, whereas Tier 1 devices only allow these functions to read data from an argument buffer. Additionally, Tier 2 devices can access more textures in an instanced draw call than Tier 1 devices. See Improving CPU performance by using argument buffers for more information about argument buffer tiers, limits, and capabilities.

This sample checks for Tier 2 argument buffer support when the renderer is initialized.

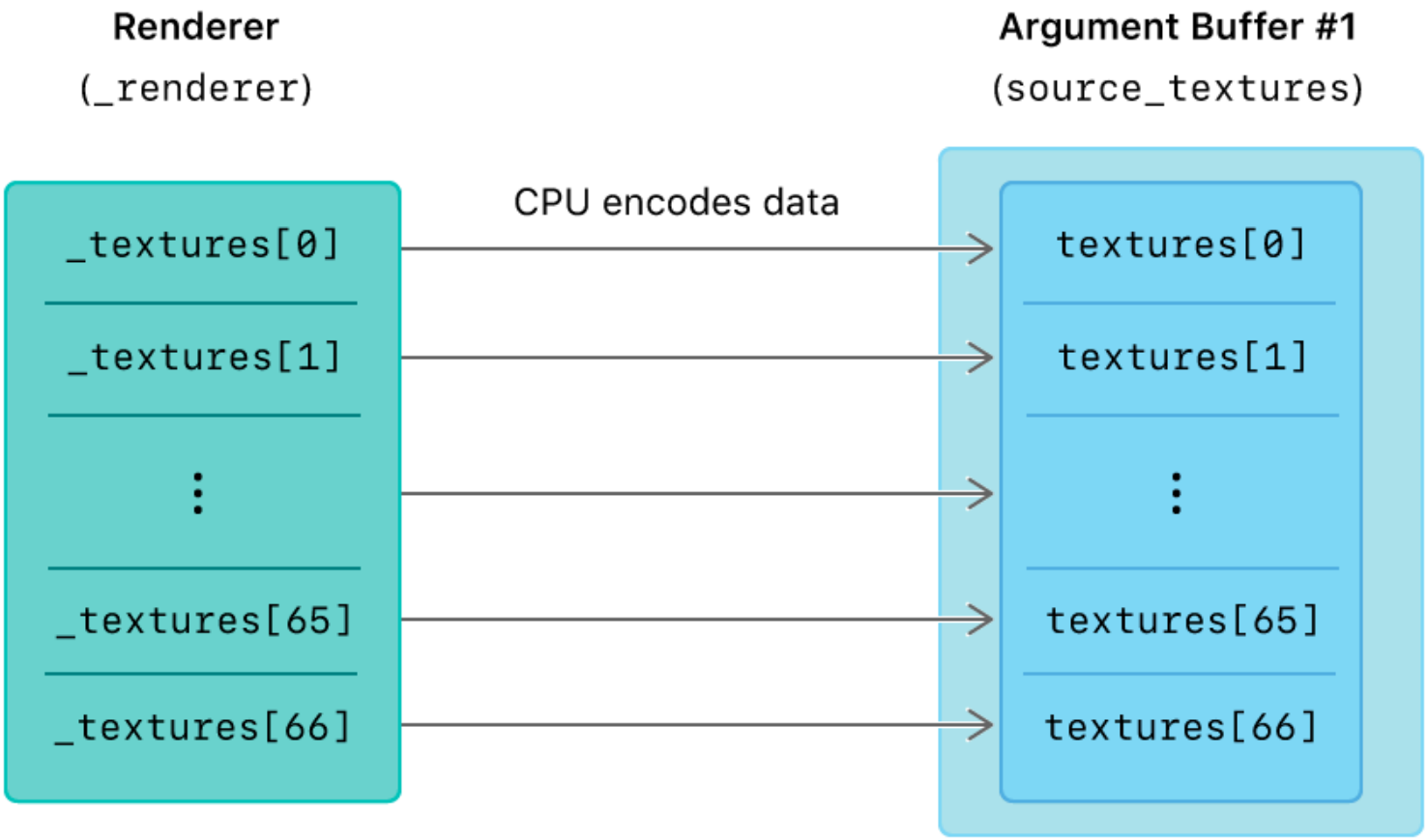
```
if(_view.device.argumentBuffersSupport != MTLArgumentBuffersTier2)
{
    NSAssert(0, @"This sample requires a Metal device that supports Tier 2 argument
}
```

Encode data into argument buffers

During initialization, the sample encodes data with the CPU into an argument buffer defined by the `SourceTextureArguments` structure.

```
struct SourceTextureArguments {
    texture2d<float>    texture [[ id(AAPLArgumentBufferIDTexture) ]];
};
```

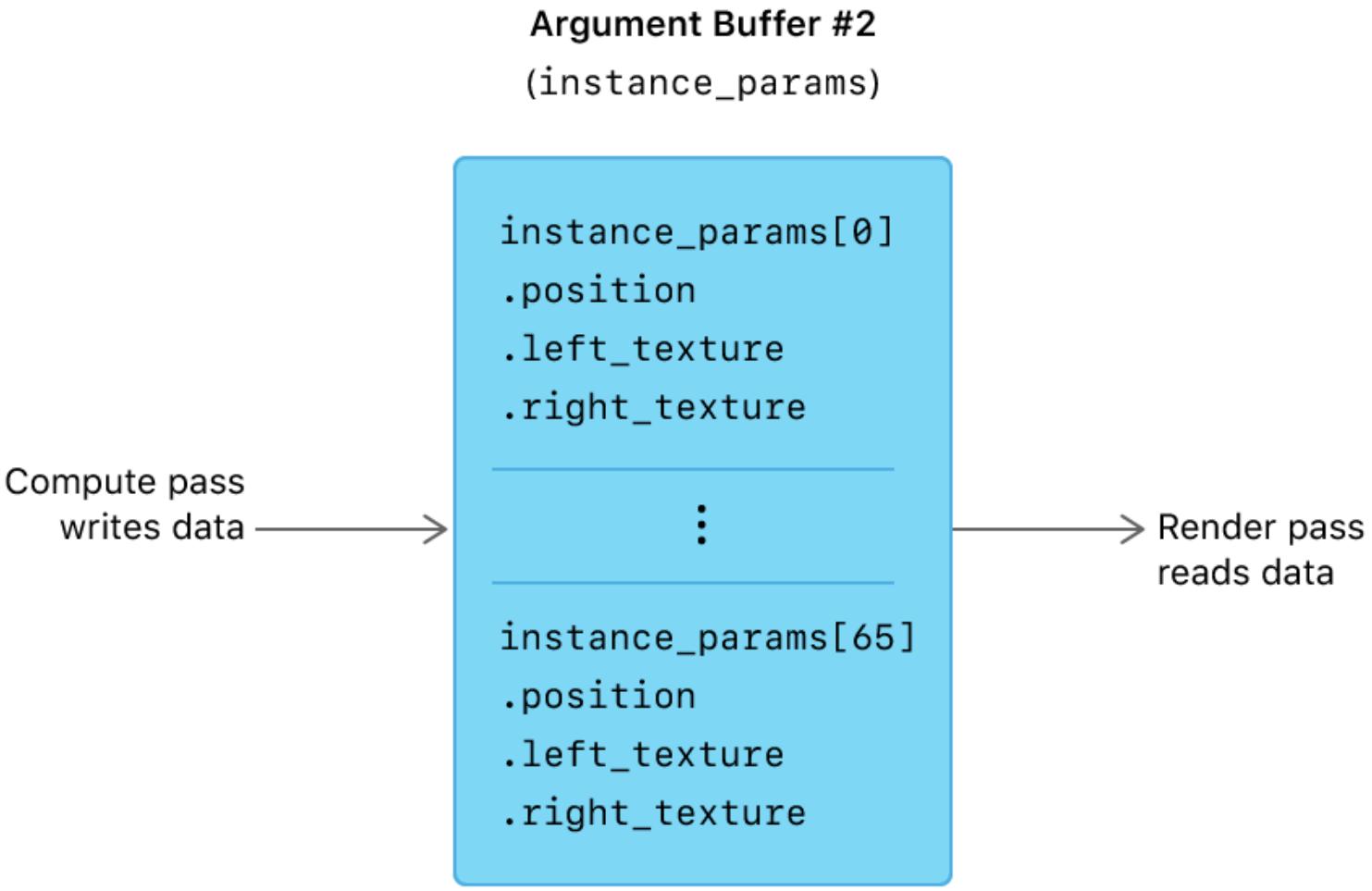
This argument buffer is backed by the `_sourceTextures` buffer and is accessed via the `source_textures` variable in the `updateInstances` function. `source_textures` is a pointer to an unbounded array of structures, each of which contains a reference to a texture.



After initialization, for each frame, the sample encodes data with the GPU into a separate argument buffer defined by the `InstanceArguments` structure.

```
struct InstanceArguments {
    vector_float2    position;
    texture2d<float> left_texture;
    texture2d<float> right_texture;
};
```

This argument buffer is backed by the `_instanceParameters` buffer and is accessed via the `instance_params` variable in the `updateInstances`, `vertexShader`, and `fragment Shader` functions. `instance_params` is an array of structures whose data is populated in a compute pass and then accessed in a render pass via an instanced draw call.



Create an array of argument buffer structures

The sample defines an `InstanceArguments` structure into which a compute function, `update Instances`, encodes a vector and two textures.

```
struct InstanceArguments {
    vector_float2    position;
    texture2d<float> left_texture;
    texture2d<float> right_texture;
```

```
};
```

Previous argument buffer samples used the `encodedLength` property to directly determine the required size for the `MTLBuffer` that backs an argument buffer structure. However, this sample needs one instance of this structure for each quad rendered by a subsequent render pass. Therefore, the sample multiplies the value of `encodedLength` by the total number of instances, which is defined by the value of the `AAPLNumInstances` constant.

```
NSUInteger instanceParameterLength = instanceParameterEncoder.encodedLength * AAPLNu

_instanceParameters = [_device newBufferWithLength:instanceParameterLength options:0
```

> **Note**
>
> The `[[id(n)]]` attribute qualifier isn't necessary to define the `InstanceArguments` structure in this sample. This qualifier is needed only when arguments are encoded with the CPU via the Metal API, and not when arguments are encoded with the GPU via a graphics or compute function.

Encode an argument buffer with a compute function

For each quad to be rendered, the sample executes the `updateInstances` compute function to determine the quad's position and textures. The compute pass executed by the sample iterates through the `instance_params` array and encodes the correct data for each quad. The sample encodes data into `instance_params` by setting `InstanceArguments` values in the array element at the `instanceID` index value.

```
// Select the element in the instance_params array which stores the parameter for th
device InstanceArguments & quad_params = instance_params[instanceID];

// Store the position of the quad.
quad_params.position = position;

// Select and store the textures to apply to this quad.
quad_params.left_texture = source_textures[left_texture_index].texture;
quad_params.right_texture = source_textures[right_texture_index].texture;
```

Render instances with an argument buffer

The sample issues an instanced draw call to render all the quads while incurring a minimal amount of CPU overhead. Combining this technique with an argument buffer allows the sample to use a unique set of resources for each quad within the same draw call, where each instance draws a single quad.

The sample declares an `instanceID` variable in both the vertex and fragment function's signatures. The render pipeline uses `instanceID` to index into the `instance_params` array that was previously encoded by the `updateInstances` compute function.

In the vertex function, `instanceID` is defined as an argument with the `[[instance_id]]` attribute qualifier.

```
vertex RasterizerData
vertexShader(uint                              vertexID        [[ vertex_id ]],
             uint                              instanceID      [[ instance_id ]],
             const device AAPLVertex          *vertices        [[ buffer(AAPLVertexBu
             const device InstanceArguments *instance_params [[ buffer(AAPLVertexBu
             constant AAPLFrameState          &frame_state     [[ buffer(AAPLVertexBu
```

The vertex function reads position data from the argument buffer to render the quad in the right place in the drawable.

```
float2 quad_position = instance_params[instanceID].position;
```

The vertex function then passes the `instanceID` variable to the fragment function, via the `RasterizerData` structure and the `[[stage_in]]` attribute qualifier. (In the fragment function, `instanceID` is accessed via the `in` argument.)

```
fragment float4
fragmentShader(RasterizerData            in              [[ stage_in ]],
               device InstanceArguments *instance_params [[ buffer(AAPLFragmentBuffe
               constant AAPLFrameState  &frame_state     [[ buffer(AAPLFragmentBuffe
```

The fragment function samples from the two textures specified in the argument buffer and then chooses an output sample based on the value of `slideFactor`.

```
texture2d<float> left_texture = instance_params[instanceID].left_texture;
texture2d<float> right_texture = instance_params[instanceID].right_texture;

float4 left_sample = left_texture.sample(texture_sampler, in.tex_coord);
float4 right_sample = right_texture.sample(texture_sampler, in.tex_coord);
```

```
if(frame_state.slideFactor < in.tex_coord.x)
{
    output_color = left_sample;
}
else
{
    output_color = right_sample;
}
```

The fragment function outputs the selected sample. The left texture slides in from the left and the right texture slides out to the right. After the right texture has completely slid off the quad, the sample assigns this texture as the left texture in the next compute pass. Thus, each texture moves from left to right across the grid of quads.

Next steps

In this sample, you learned how to encode resources into argument buffers with a graphics or compute function. In Rendering terrain dynamically with argument buffers, you'll learn how to combine several argument buffer techniques to render a dynamic terrain in real time.

See Also

Argument buffers

- Improving CPU performance by using argument buffers
  Optimize your app's performance by grouping your resources into argument buffers.
- Managing groups of resources with argument buffers
  Create argument buffers to organize related resources.
- Tracking the resource residency of argument buffers
  Optimize resource performance within an argument buffer.
- Indexing argument buffers
  Assign resource indices within an argument buffer.
- Rendering terrain dynamically with argument buffers
  Use argument buffers to render terrain in real time with a GPU-driven pipeline.

{} Using argument buffers with resource heaps

Reduce CPU overhead by using arrays inside argument buffers and combining them with resource heaps.

`class MTLArgumentDescriptor`

A representation of an argument within an argument buffer.

`protocol MTLArgumentEncoder`

An interface you can use to encode argument data into an argument buffer.

`let MTLAttributeStrideStatic: Int`