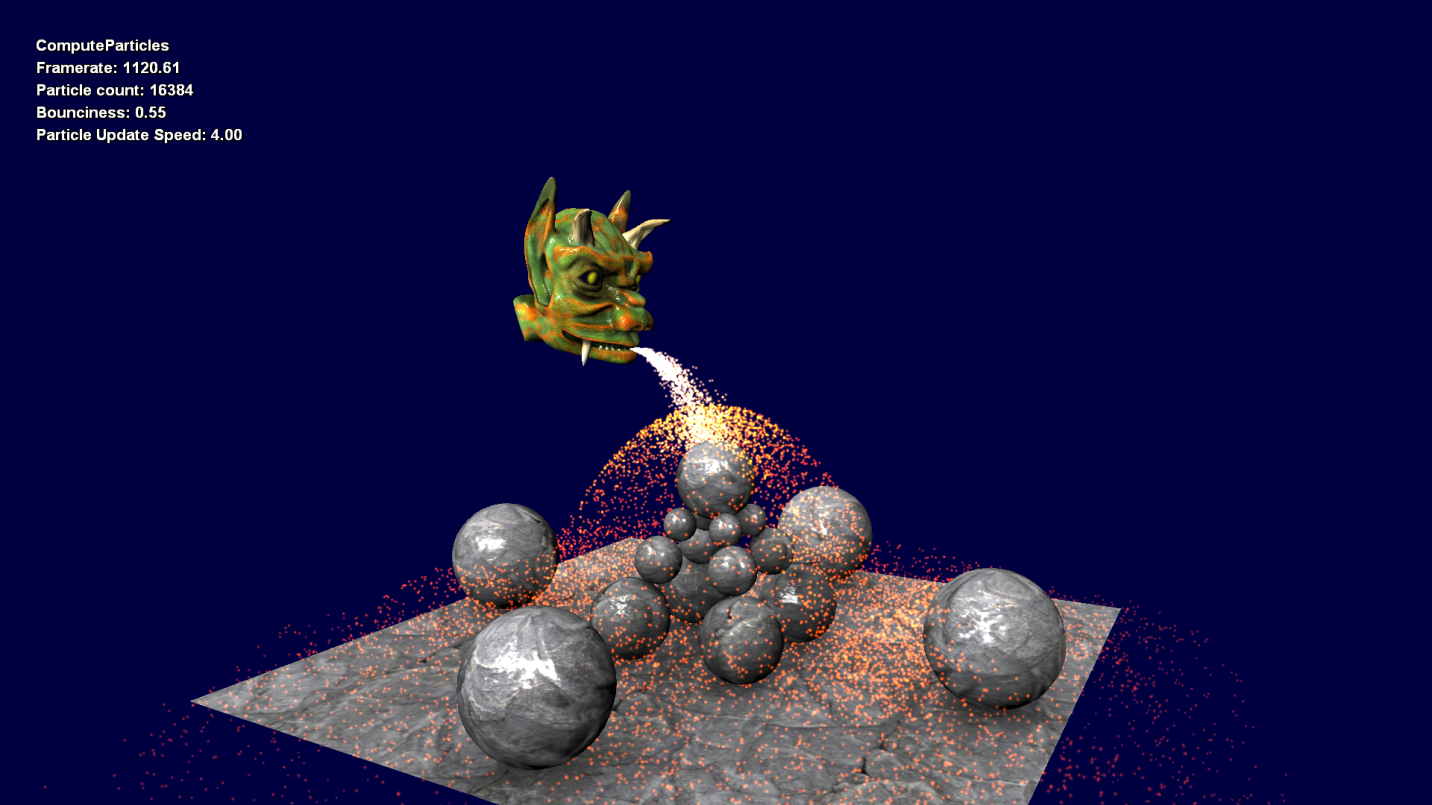


Compute Particles

*This sample is compatible with the Microsoft Game Development Kit (October 2021)*

# Description

This sample demonstrates how to use compute shaders and append buffers to perform a basic particle simulation and performantly render a silly number of particles.



# Building the sample

If using an Xbox One devkit, set the active solution platform to Gaming.Xbox.XboxOne.x64.

If using Xbox Series X|S devkit, set the active solution platform to Gaming.Xbox.Scarlett.x64.

*For more information, see* Running samples*, in the GDK documentation.*

# Using the sample

This sample uses the following controls.

|  |  |
| --- | --- |
| Action | Gamepad |
| Exit the sample. | Select |
| Increase/Decrease particle bounciness | Right/Left Trigger |
| Rotate camera | Left & Right Stick |
| Move particle emitter | Left & Right Stick + Right Shoulder |
| Toggle particle rendering | A Button |
| Toggle particle update | B Button |
| Increase/Decrease number of particles | D-Pad Up/Down |

# Implementation notes

This sample demonstrates some of the more esoteric and interesting techniques available with D3D11 and Compute Shaders. There are three parts of interest to this sample. The first two are related to how particles are updated and culled, and the third is how the result of the Compute Shader stage is consumed by the regular rendering pipeline.

1. **Particle Simulation**

The AdvanceParticlesCS computer shader simulation phase contains two main steps. First, particle positions, velocities, and age are read from a UAV buffer and simulated in world-space. Then collisions against simplified world geometry (ground plane and spheres) are calculated using a brute-force approach. The new positions, velocities, and ages are written back into the same UAV buffer, overwriting the previously read data.

1. **Particle Cull and Write**

A simple plane culling algorithm is applied to each particle to consider whether it is visible within the view frustum. When a particle is visible, its position is appended to an append buffer for rendering.

An append buffer can be used by creating a Unordered Access View (UAV) for a buffer resource with the ID3D12Device::CreateUnorderedAccessView(…) API, specifying a second resource as the ‘pCounterResource’ parameter. The counter resource must be at least 4 bytes to store the append buffer’s current count (one 32-bit unsigned integer.) A UAV is also created for the counter resource which allows us to clear the count each frame with the ID3D12GraphicsCommandList::ClearUnorderedAccessViewUint(…) API. Once created we simply bind the buffer to a UAV shader slot declared as an AppendStructuredBuffer<…>.

The AdvanceParticlesCS compute shader simulates and adds active particle instances to the append buffer. Once complete, the ID3D12GraphicsCommandList::CopyBufferRegion(…) API is used to copy the particle count from our counter resource to a indirect argument buffer resource, which can be used as an input to the ID3D12GraphicsCommandList::ExecuteIndirect(…) API. This allows us to draw only visible particles dictated by the AdvanceParticlesCS’s frustum-cull tests.

1. **Rendering**

The ID3D12GraphicsCommandList::ExecuteIndirect(…) API is used to dispatch particle rendering. A command signature, created used ID3D12Device::CreateCommandSignature(…), is required to specify which type of command ExecuteIndirect will dispatch. The command type determines how the contents of the indirect argument buffer will be interpreted. In our case the command type is ‘Draw’, which correlates to the ID3D12GraphicsCommandList::DrawInstanced(…) API – four 32-bit unsigned integers. We hardcode VertexCountPerInstance to 4, and copy the particle count to the InstanceCount location each frame.

The vertex attributes are hardcoded as a constant lookup table in the vertex shader. The Vertex ID (SV\_VertexID) is used to index into this lookup table to access each vertex’s attributes. The Instance ID (SV\_InstanceID) is used to index into the particle instance buffer to access each instance’s properties.

# Update history

March 2019 – Port to new template from legacy Xbox Sample Framework.

# Privacy Statement

When compiling and running a sample, the file name of the sample executable will be sent to Microsoft to help track sample usage. To opt-out of this data collection, you can remove the block of code in Main.cpp labeled “Sample Usage Telemetry”.

For more information about Microsoft’s privacy policies in general, see the [Microsoft Privacy Statement](https://privacy.microsoft.com/en-us/privacystatement/).