Efficient instancing in a streaming scenario

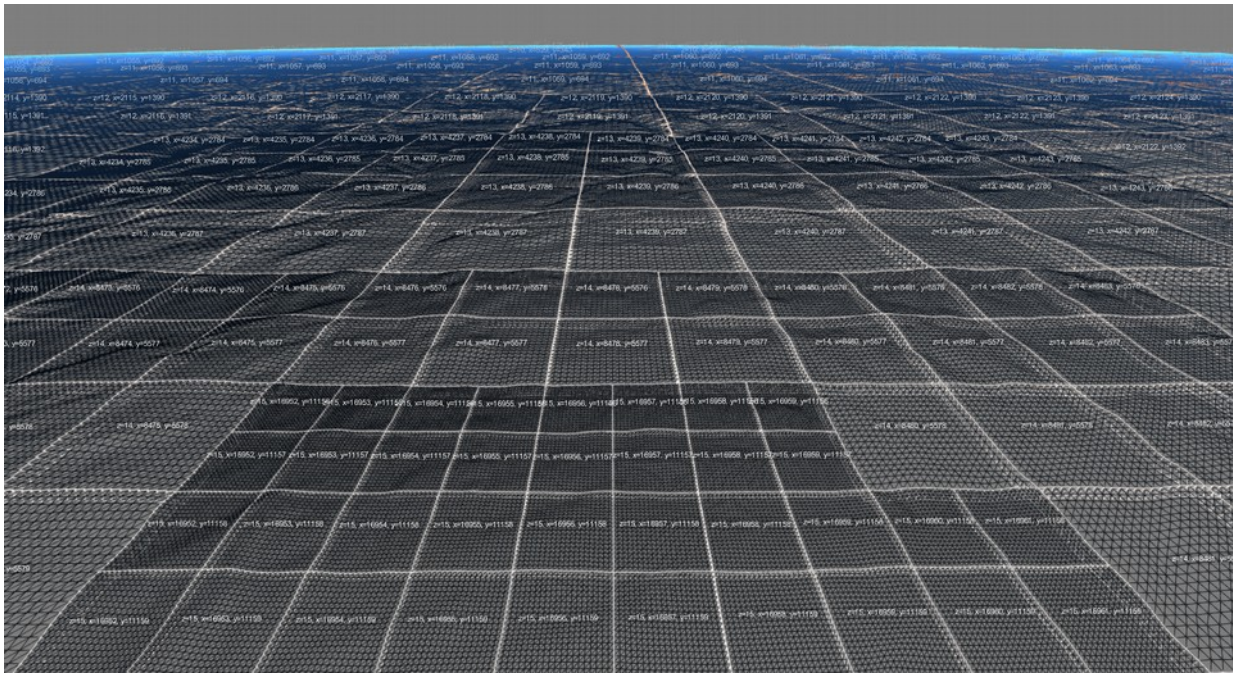
Author: Serge@BitBunch.eu

URL: MilitaryOperationsHQ.com

Table of Contents

The problem

You're building a game-world that is big, so big in a fact that not all of it can be loaded into memory at once. You also don't want to introduce portals or level loading. You want the player to have a uninterrupted experience.



For true continuous streaming, a typical scenario would be something like this:

- The world is partitioned into tiles (Quad-tree)
- When the Camera moves, tile-data is read from disk and pre-processed in the back-ground.
- We need to render meshes for each tile.
- There can be more than 1000 tiles in the AOI, more than 100 different meshes and up to 10000 instances per mesh on one tile.

How to improve from worst-case 1000000000 draw calls to best-case 1 draw call?

Introduction

To focus on the render-data preparation specifically, I assume the reader is familiar with the following concepts:

- Instanced mesh rendering
- Compute shaders
- AOI (Area Of Interest)
- Quad-tree tile-based space partitioning

For an introduction I recommend this BLOG entry on our website: (http://militaryoperationshq.com/dev-blog-a-global-scope/)

I will use OpenGL to demonstrate details because we use it ourselves and because it is the platform independent alternative. The technique however can be adapted for any modern graphics API that supports compute shaders.

The solution

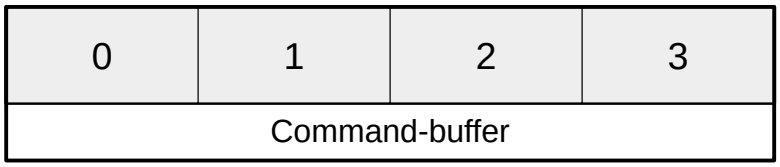
The solution is to do the work on the GPU. This is the type of processing a GPU is particularly good at.

The diagrams below show memory layout.
Each colour represents a different type of instance data, stored non-interleaved. For example, position, texture-array layer-index or mesh scale-factor etc.
Within each instance-data-type (colour) range, a sub-range (grey) will be used for storing data for instances of a particular mesh. In this example there are 4 different meshes that can be instanced. Within the sub-range, there is room to store instance-data for "budget" amount of instances. After loop-stage step 4, we know exactly where to store instance data of each type (pos, tex-index, scale, etc.) for a particular mesh-type. In this example, the scene contains no mesh-type 2 instances and many mesh-type 3 instances.

Prepare once at start-up

- Load all mesh data of the models you want to be able to show in one buffer.
- Prepare GL state by creating a Vertex Array Object containing all bindings.

- Create a command-buffer containing Indirect-Structures, one structure for each mesh that you want to be able to render.



- Fill the Indirect-Structure members that point to (non-instance) mesh vertex data.

Steps for one new tile entering the AOI

1. Read geometry from disk
2. Rasterize geometry into a material-map
3. Generate instance-points covering the tile. Select a grid-density and randomise points inside their grid-cell to make it look natural if you're doing procedural instancing. Whole papers have been written about this topic alone.
4. Sample from the material-map at the grid-point to cull points and decorate data. Store the result in a buffer per tile.
5. Keep the result-buffer of a tile for as long as it is in the AOI

Step 1, 2, 3 and 4 may well be replaced by simply loading points from disk if they are pre-calculated offline. In our case we cover the entire planet, so we need to store land-use data in vector form and convert it into raster data online, to keep the install size manageable.

Steps for each loop

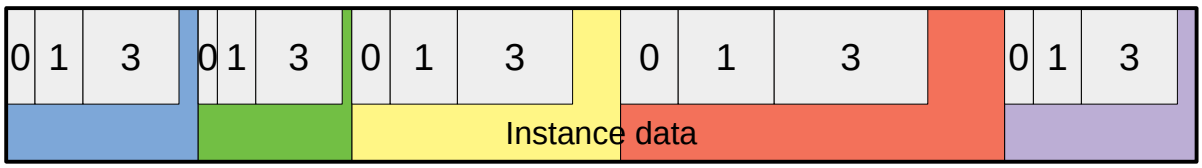
This is where things get interesting.

1. Do frustum and other culling of the tiles so you know what tiles are visible and contain meshes that need rendering.

2. Clear instance-count and base-instance fields of indirect-structures in the command-buffer. Run a simple compute shader for this. If you would map the buffer or use *glBufferData* to allow access from the CPU, you introduce an expensive upload and synchronisation which we want to prevent.

3. Run a compute shader over the tile-set in view to determine which meshes to render. Just count instances per mesh in the instance-count member of the Indirect_structure.
   This may require sampling from the material map again or doing other calculations to pick a mesh LOD or reflect game-state. It may very well require procedural math to "randomly" spawn meshes. This all depends on your particular game requirements.

4. Fill-in the base-instance member of the Indirect-Structures by launching a single compute shader instance.



5. Run a compute shader to prepare render data. Do the calculations that determine what mesh to select, again. Claim a slot in the vertex-attributes buffer and store render data. since at this point we already know exactly how many instances of each mesh will need rendering (all counts and offsets), we know in what range a particular mesh instance needs to store instance-data. The order within the range for a particular mesh doesn't matter.

The important, maybe counter intuitive thing here is, that we do all calculation to determine what mesh to instance, twice. We don't store the results from the first time. It would be complicated, memory consuming and slow to remember what mesh instance of what tile ends-up at what vertex-data location in the render buffer, just so we can look up an earlier stored result. It may feel wasteful to do the same work twice, but that is procedural thinking. On the GPU it is often faster to recalculate something then to store and read back an earlier result.
Now everything is done on the GPU and we only need to add some memory-barriers to make sure data is actually committed before a next step is using it.

Atomic operations

Note that step 3 and 5 of the loop-stage require the compute shader to use atomic operations. They are guaranteed to not conflict with other shader-instances when writing to the same memory location.

Instance budget

You need to select a budget for the maximum number of meshes that can be drawn at once. It defines the size of the instance-data buffer. This budget may not cover certain extreme situations. This means we need to make sure we do not exceed the budget.
Step 4 updates the base-instance of the indirect-structure. At that point we can detect if we exceed the budget. We can simply force instance-counts to zero when we reach the budget. But this will have the effect of potentially very visible mesh instances to be in or excluded from the render-set each loop.
To solve this, sort the indirect-structures, representing meshes, in the command-buffer from high to low detail. This is only needed once at start-up.
That way the first meshes that will be dropped are low LOD and should have the least impact.
If you're using tessellation to handle LOD, you'll have to solve this differently or make sure your budget can handle the extreme cases.

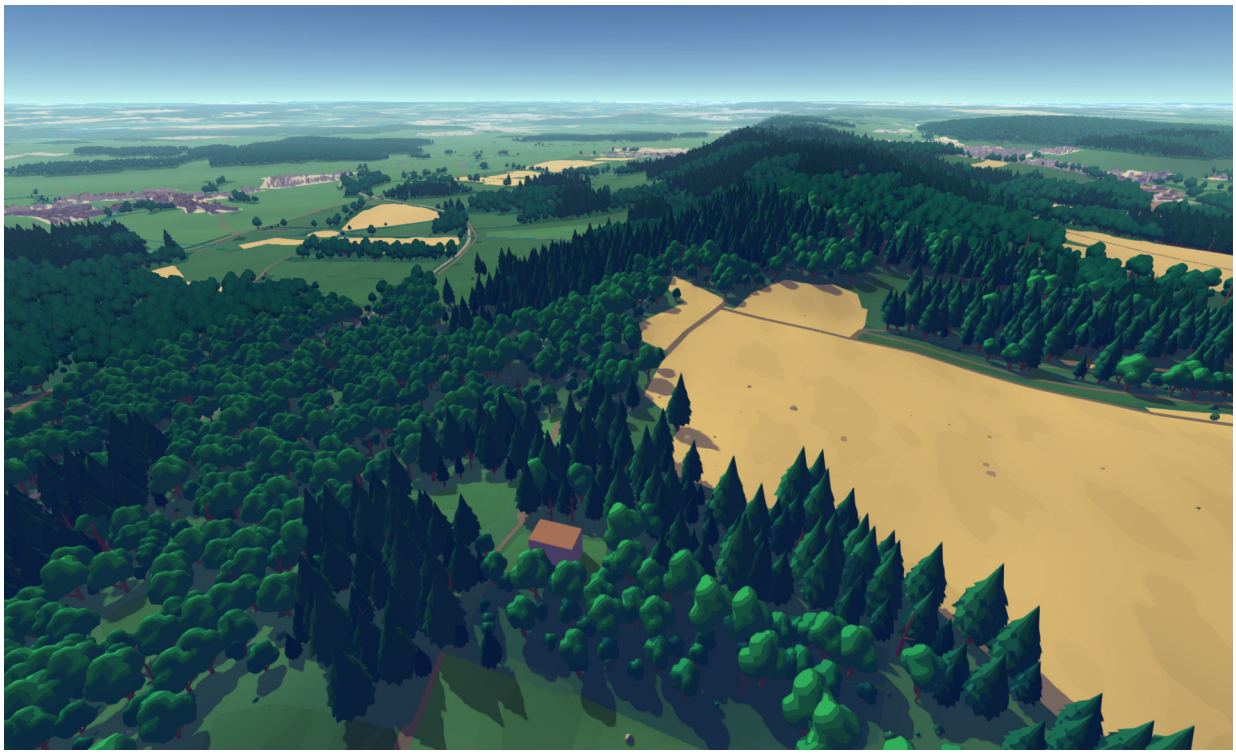
Ready to render

We now have one buffer containing all render data needed to render all instances of all meshes on all tiles, in one call.
We simply do a single Render call using the buffer that contains the indirect-structures. In fact, we render *all* possible meshes. If, for the current situation, some meshes do not need to be rendered, their instance-count in the indirect-structure will be zero and it will be skipped with very little to no overhead.

How it used to be

In a traditional scenario we may have filled a indirect-structure buffer with only structures for the meshes that need rendering. Then copying all render data in a vertex-attribute buffer, in an order to match the order of indirect-structures in the command-buffer. Which means we need a sorting step. Next, an upload of this render data to the GPU is required. Since the upload is slow, we will probably need to double or, even better, triple buffer this to hide transfer and driver stages so we don't end up waiting before we can access/render a buffer.

Summary



Key points

- Preprocess and store intermediate data on the gpu
- Make mesh instance render order fixed and render all meshes always
- Use a 2 pass approach, first count instances so we can predict memory layout the second time.

Benefits

- No upload of render data each render-loop
- No need to refill/reorder the command-buffer (indirect-structures) every loop
- No sort needed to pack vertex-data for mesh instances
- No need for double/triple buffering

Improvements

The most performance is gained by selecting the best design for your software implementation. Nevertheless, there is some low hanging fruit to be picked before going into proper profiling and tackling bottlenecks.

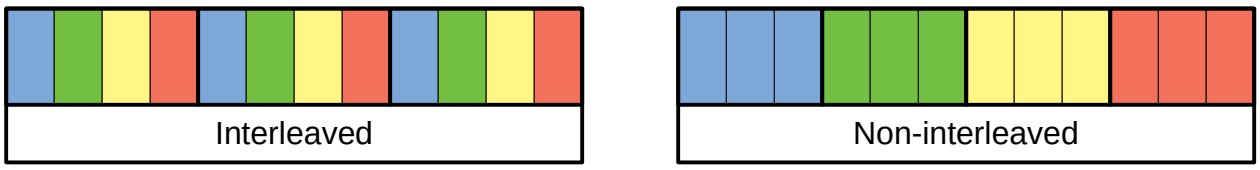
Memory allocation

You should use glBufferStorage to allocate GPU memory for buffers. Since we never need to access GPU memory from the CPU, we can do this:

```
glBufferStorage(GL_ARRAY_BUFFER, data_size, &data[0], 0);
```

The last parameter tells the GL how we want to access the memory.
In our case we simple pass 0 meaning we will only access it from the GPU.
This can make a substantial difference depending on vendor, platform and driver.
It allows the implementation to make performance critical assumptions when allocating GPU memory.

Storing data



This article describes vertex-data that is stored non-interleaved. We are in massively parallel country now, not OO. Without going into details, the memory access patterns make it more efficient if, for example, all positions of all vertices/instances are packed. The same goes for all other attributes.

Frustum culling

In this example, tile frustum-culling is done on the CPU. A tile however may contain many mesh instances and it makes good sense to do frustum culling for those as well. It can be easily integrated into step 3 and performed on the GPU.

Launching compute shaders

The pseudo-code example shows how compute shaders are launched for the number of points on each tile. This means the number of points needs to be known on the CPU. Even though this is determined in the background, downloading this information requires an expensive transfer/sync.
We can store this information on the GPU and use a *glDispatchComputeIndirect* call that reads the launch size from GPU memory.

A trend

This article shows how more work can be pushed to the GPU.
We took this notion to the extreme by designing an engine from the ground up, that is completely running on the GPU. The CPU is only doing I/O, user interaction and starting GPU jobs.
You can read more about our "Metis Tech" on our blog-page:
(http://militaryoperationshq.com/blog/)

The main benefits are the lack of large data up/down-loads and profiting from the huge difference in processing power between GPU and CPU.
At some point this gap will become a limiting bottleneck. According to NVidia, the GPU is expected to be 1000x more powerful than the CPU by 2025!
(https://www.nextplatform.com/2017/05/16/embiggening-bite-gpus-take-datacenter-compute/)

Appendix A - Pseudo C++ code

```
//! \brief Prepare render data to draw all static meshes in one draw call
void prepare_instancing( const std::vector<int32_t>& p_tiles_in_view // Points per tile
                       , const std::vector<GLuint> p_point_buffer
                       , int32_t p_mesh_count
                       , GLuint p_scratch_buffer
                       , GLuint p_command_buffer
                       , GLuint p_render_buffer
                       , GLuint p_clear_shader
                       , GLuint p_count_shader
                       , GLuint p_update_shader
                       , GLuint p_prepare_shader)
{
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, p_command_buffer);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, p_scratch_buffer);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 3, p_render_buffer);

    // 2. Clear instance base and count
    glUseProgram(p_clear_shader);
    glDispatchCompute(p_mesh_count, 1, 1);
    glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);

    // 3. Count instances per mesh
    glUseProgram(p_count_shader);
    for (int32_t l_tile_index = 0; l_tile_index < p_tiles_in_view.size(); ++l_tile_index)
    {
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, p_point_buffer[l_tile_index]);
        glDispatchCompute(p_tiles_in_view[l_tile_index], 1, 1);
        glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
    }

    // 4. Update instance base
    glUseProgram(p_update_shader);
    glDispatchCompute(1, 1, 1);
    glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);

    // 5. Prepare render data
    glUseProgram(p_prepare_shader);
    for (int32_t l_tile_index = 0; l_tile_index < p_tiles_in_view.size(); ++l_tile_index)
    {
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, p_point_buffer[l_tile_index]);
        glDispatchCompute(p_tiles_in_view[l_tile_index], 1, 1);
        glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
    }
    glUseProgram(0);
    glBindBuffersBase(GL_SHADER_STORAGE_BUFFER, 0, 4, nullptr);
}

//! \brief Render all instances of all meshes on all tiles in one draw call
void render_instanced( GLuint p_vao
                     , GLuint p_command_buffer
                     , GLuint p_render_shader
                     , int32_t p_mesh_count)    // Number of different meshes that can be shown
{
    glBindVertexArray(p_vao);
    glUseProgram(p_render_shader);
    glBindBuffer(GL_DRAW_INDIRECT_BUFFER, p_command_buffer);
    glMultiDrawElementsIndirect(GL_TRIANGLES, GL_UNSIGNED_INT, 0, p_mesh_count, 0);
    glBindBuffer(GL_DRAW_INDIRECT_BUFFER, 0);
    glUseProgram(0);
    glBindVertexArray(0);
}
```

Appendix B – Compute-shader pseudo-code

2. Clear shader

```
//********************************************************************************
//! \brief 2. p_clear_shader: Clear counts and offsets
//********************************************************************************
// The local workgroup size
layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

// Input: Contains the indirects-structs for rendering the meshes
layout(std430, binding=0) buffer p_command_buffer
{
    uint p_indirect_structs[];
};

// IO: Containing uints for counting point-instance-data per mesh, and claiming slots
layout(std430, binding=2) buffer p_scratch_buffer
{
    uint p_instance_counts[];    // Globally for all tiles. Size = number of mesh variants
};

void main()
{
    uint l_invocation_id = gl_GlobalInvocationID.x;
    p_indirect_structs[l_invocation_id * 5 + 1] = 0; // 5 uints, second is the instance-count.
    p_instance_counts[l_invocation_id] = 0;
}
```

3. Count shader

```
//********************************************************************************
//! \brief 3. p_count_shader: Count instances
//********************************************************************************
// The local workgroup size
layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

// Output: Contains the indirect-structs for rendering the meshes
layout(std430, binding=0) buffer p_command_buffer
{
    uint p_indirect_structs[];   // Globally for all tiles
};

layout(std430, binding=1) buffer p_point_buffer
{
    uint p_point_data[];
};

void main()
{
    uint l_invocation_id = gl_GlobalInvocationID.x;

    //! \note What p_point_data contains is application specific. Probably at least a tile-local position.
    uint l_data = p_point_data[l_invocation_id];

    //! \todo Use data in p_point_data to determine which mesh to render, if at all.
    uint l_mesh_index = 0;
    atomicAdd(p_indirect_structs[l_mesh_index], 1); // Count per instance
}
```

4. Update shader

```
//********************************************************************************
//! \brief 4. p_update_shader: Update instance base
//********************************************************************************
// The local workgroup size

layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

// Input: Contains the indirect-structs for rendering the meshes
layout(std430, binding=0) buffer p_command_buffer
{
    uint p_indirect_structs[];
};

uniform uint g_indirect_struct_count = 0;
uniform uint g_instance_budget = 0;

void main()
{
    uint l_invocation_id = gl_GlobalInvocationID.x;

    // This compute-shader should have been launched with 1 global instance!
    if (l_invocation_id > 0)
    {
        return;
    }

    //  Update base-instance values in DrawElementsIndirectCommand
    int l_index, l_n = 0;
    p_indirect_structs[l_index * 5 + 4] = 0;  //  First entry is zero

    bool l_capacity_reached = false;
    for (l_index = 1; l_index < g_indirect_struct_count; ++l_index)
    {
        l_n = l_index - 1; // Index to the indirect-struct before

        uint l_base_instance = p_indirect_structs[l_n * 5 + 4] + p_indirect_structs[l_n * 5 + 1];

        // If the budget is exceeded, set instance count to zero
        if (l_base_instance >= g_instance_budget)
        {
            p_indirect_structs[l_index * 5 + 1] = 0;
            p_indirect_structs[l_index * 5 + 4] = p_indirect_structs[l_n * 5 + 4];
        }
        else
        {
            p_indirect_structs[l_index * 5 + 4] = l_base_instance;
        }
    }
}
```

5. Prepare shader

```
// The local workgroup size
layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

// Input: Contains the indirect-structs for rendering the meshes
layout(std430, binding=0) buffer p_command_buffer
{
    uint p_indirect_structs[];
};
// Input: Containing point data
layout(std430, binding=1) buffer p_point_buffer
{
    uint p_point_data[];
};
// IO: Containing mesh-counts for claiming slots
layout(std430, binding=2) buffer p_scratch_buffer
{
    uint p_instance_counts[];    // Globally for all tiles. Size = g_indirect_struct_count
};
// Output: Containing render data
layout(std430, binding=3) buffer p_render_buffer
{
    uint p_render_data[];
};

uniform uint g_indirect_struct_count = 0;
uniform uint g_instance_budget = 0;

void main()
{
    uint l_invocation_id = gl_GlobalInvocationID.x;
    uint l_data = p_point_data[l_invocation_id];

    //! \todo Use data in p_point_data to determine which mesh to render, if at all. Again.
    uint l_mesh_index = 0;

    // This should never happen!
    if ( l_mesh_index >= g_IndirectStructCount)
    {
        return;
    }

    // Only process meshes that have an instance count > 0
    if (p_indirect_structs[l_mesh_index * 5 + 1] == 0)
    {
        return;
    }

    // Reserve a spot to copy the instance data to
    uint l_slot_index = atomicAdd(p_instance_counts[l_mesh_index], 1);

    // From mesh-local to global instance-index
    l_slot_index += p_indirect_structs[l_mesh_index * 5 + 4];

    // Make sure to not trigger rendering for more instances than there is budget for.
    if (l_slot_index >= g_instance_budget)
    {
        return;
    }

    //! \todo Write any data you prepare for rendering to p_render_data using l_slot_index
}
```