Unity

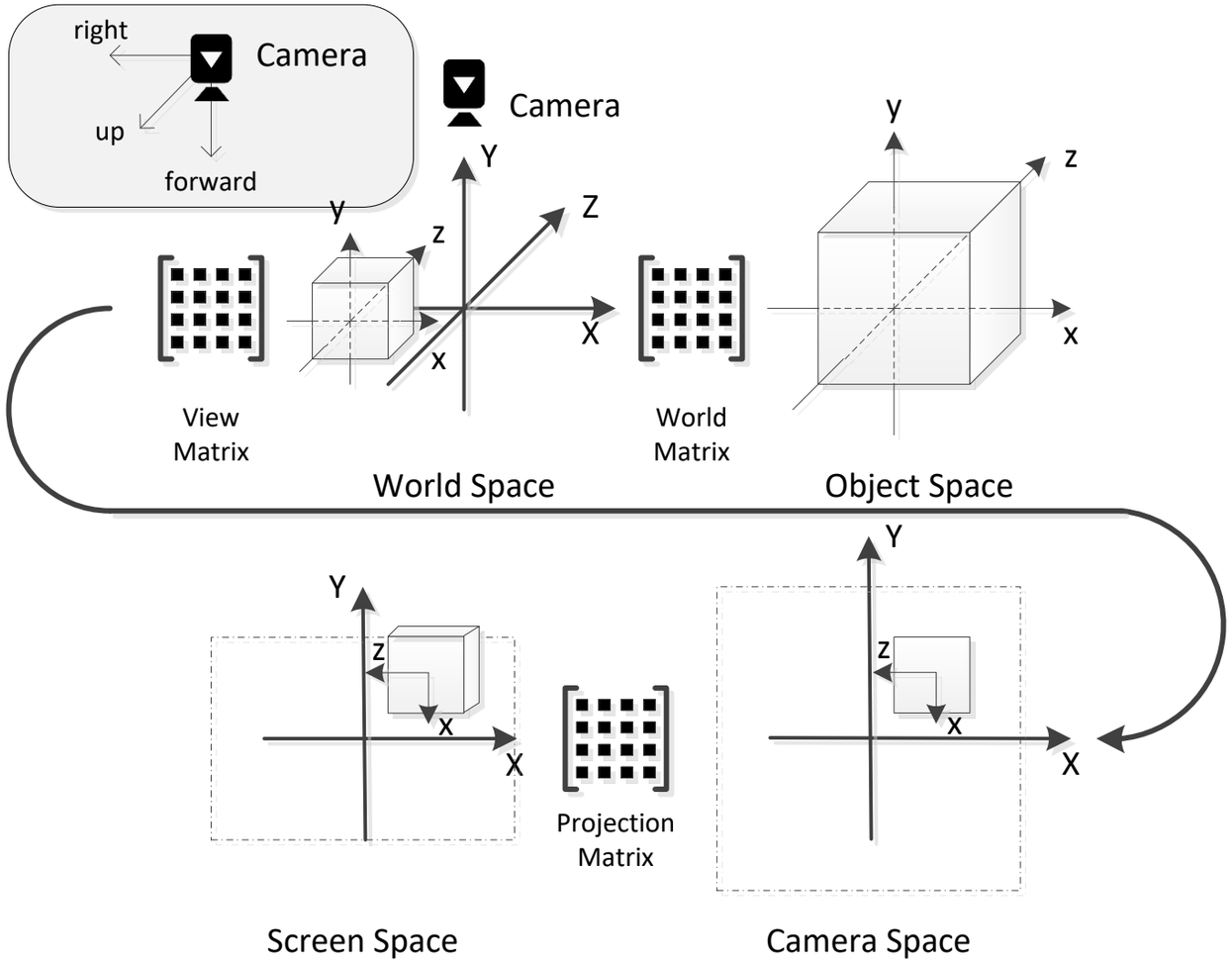
Optimizing Graphics Performance

jintaeks@dongseo.ac.kr

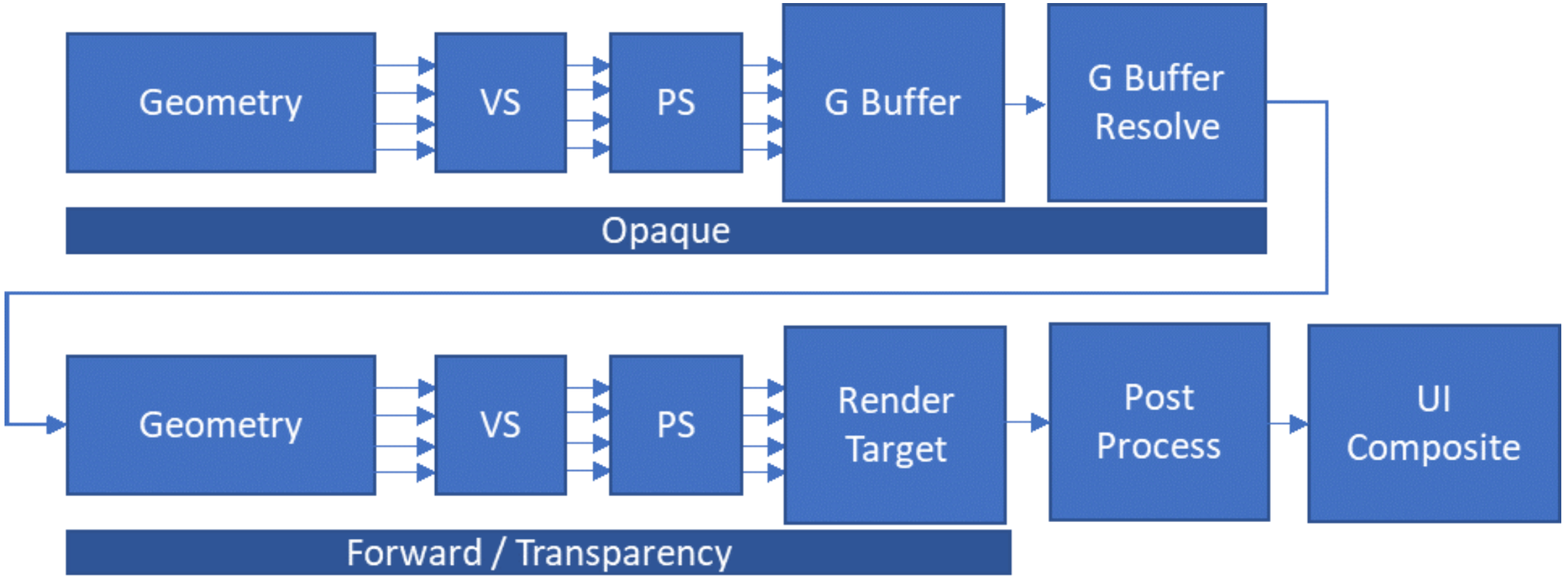
August 23th, 2019

DSU Dongseo University 동서대학교

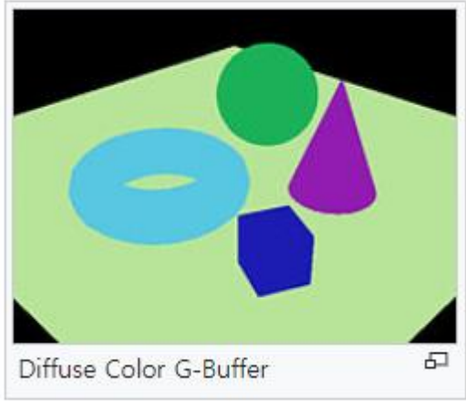
Graphics Pipeline



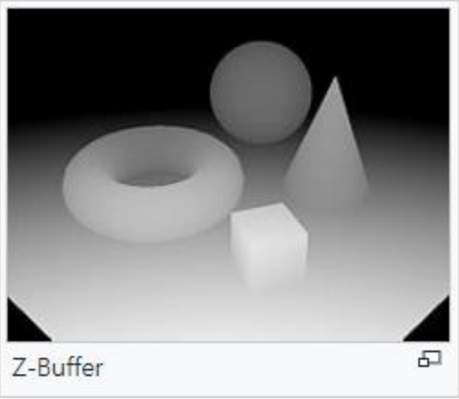
Post Processing



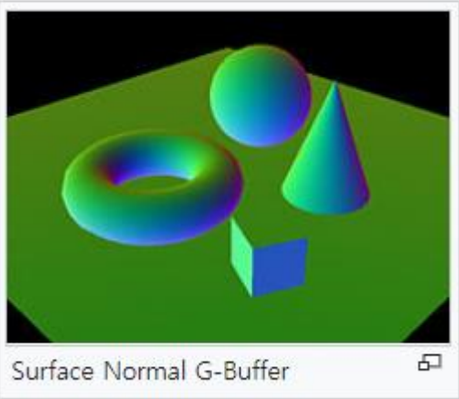
Deferred Rendering



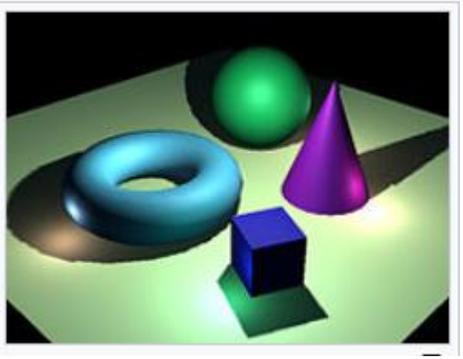
+



+



=



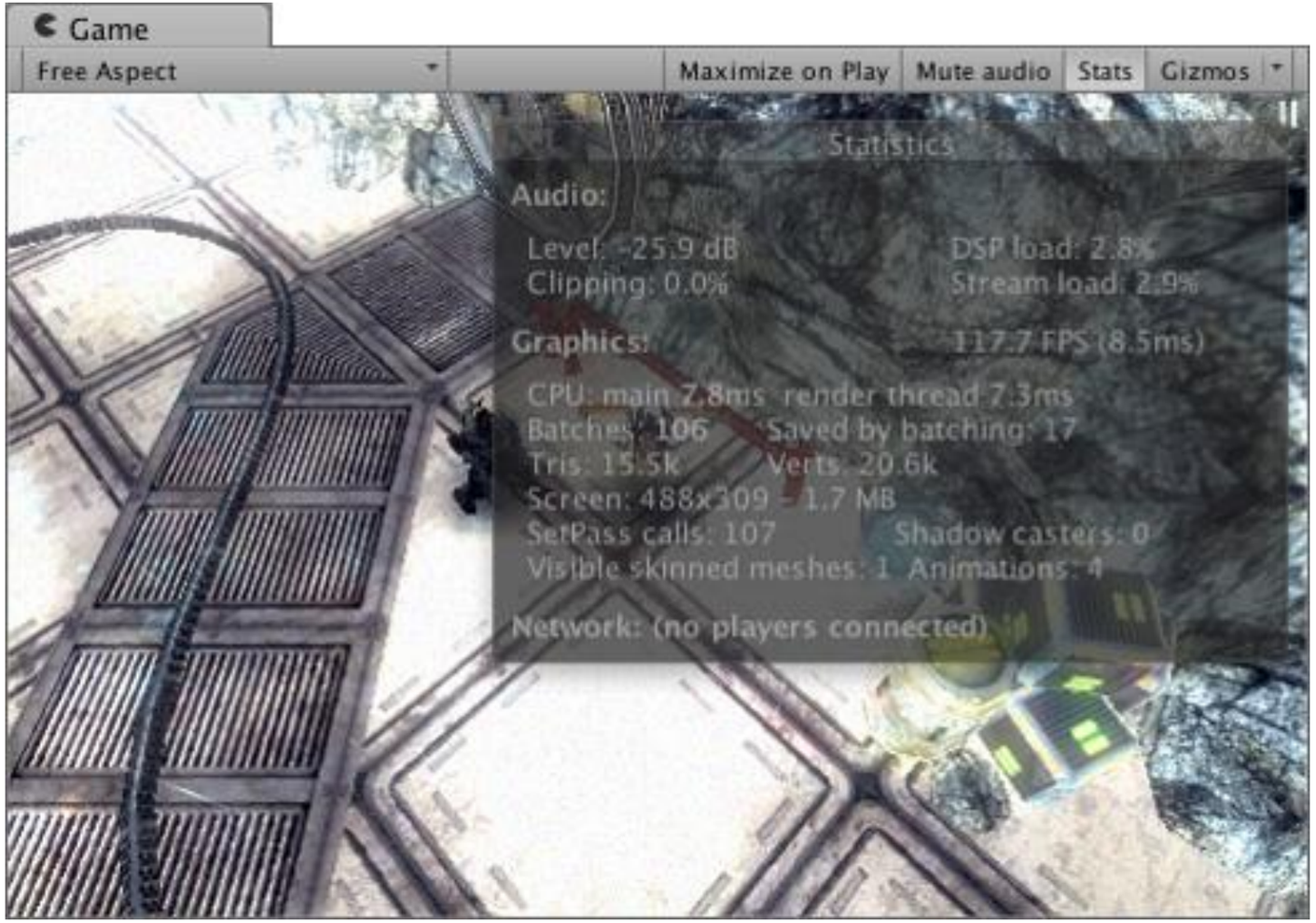
Locate high graphics impact

- The graphical parts of your game can primarily impact on two systems of the computer: the GPU and the CPU.
- The first rule of any optimization is to *find **where** the performance problem is*, because strategies for optimizing for *GPU vs. CPU* are quite different.

Common bottlenecks

- GPU is often limited by ***fillrate*** or memory bandwidth.
  - Lower the display resolution and run the game.
  - If a lower display resolution makes the game run faster, you may be limited by fillrate on the GPU.
- CPU is often limited by *the number of batches that need to be rendered*.
  - Check “batches” in the Rendering Statistics window.
  - The more batches are being rendered, the higher the cost to the CPU.

Stats on GameView



Less-common bottlenecks:

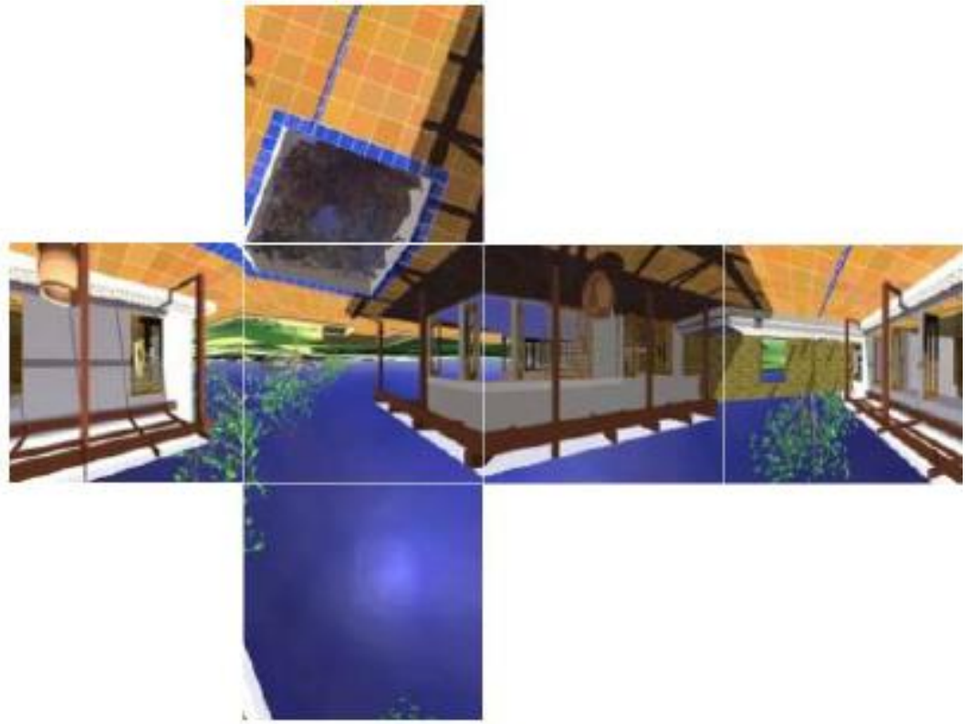
- The GPU has too many vertices to process.
  - Generally speaking, aim for no more than 100,000 vertices on mobile.
  - A PC manages well even with *several million vertices*, but it is still good practice to keep this number as low as possible through optimization.
- The CPU has too many vertices to process.
  - This could be in *skinned meshes, cloth simulation, particles, or other game objects and meshes*.
- If rendering is not a problem on the GPU or the CPU, there may be an issue elsewhere - for example, in your script or physics.
  - Use the *Unity Profiler* to locate the problem.

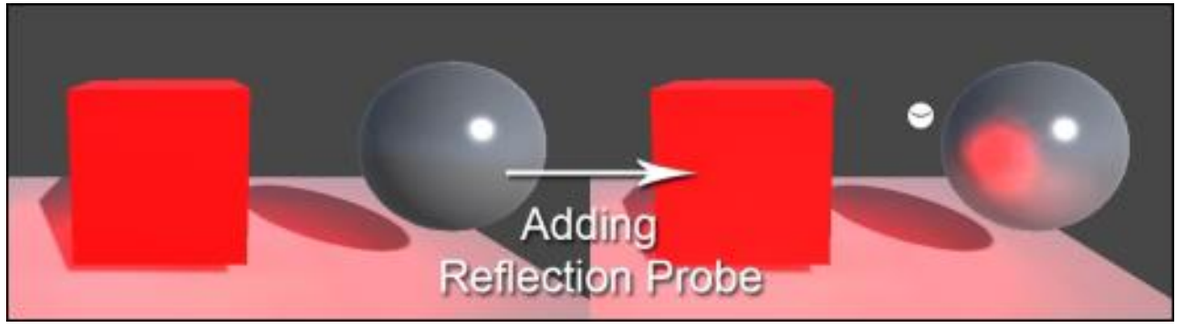
CPU Optimization

Reduce the visible object count.

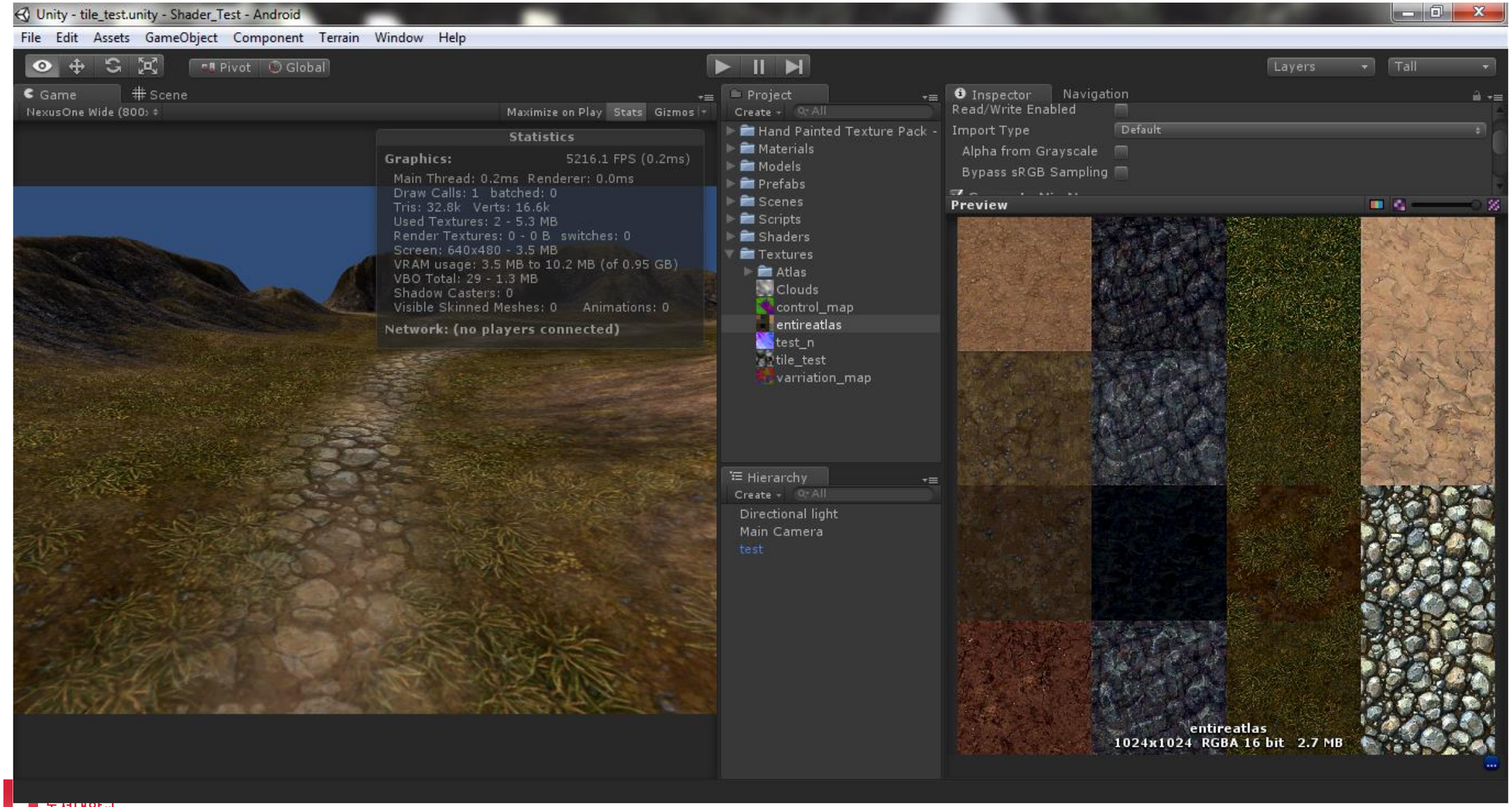
- ❑ *Combine close objects together*, either manually or using Unity's draw call batching.
- ❑ Use *fewer materials* in your objects by putting separate textures into a larger *texture atlas*.
- ❑ Use fewer things that cause objects to be rendered multiple times (such as *reflections, shadows and per-pixel lights*).

Do not use real-time reflection





Use texture atlas



GPU: Optimizing Model geometry

There are two basic rules for optimizing the geometry of a Model:

- ❑ Don’t use any more triangles than necessary.
- ❑ Try to keep the number of UV mapping seams and hard edges (doubled-up vertices) as low as possible.

Modeling characters for optimal performance

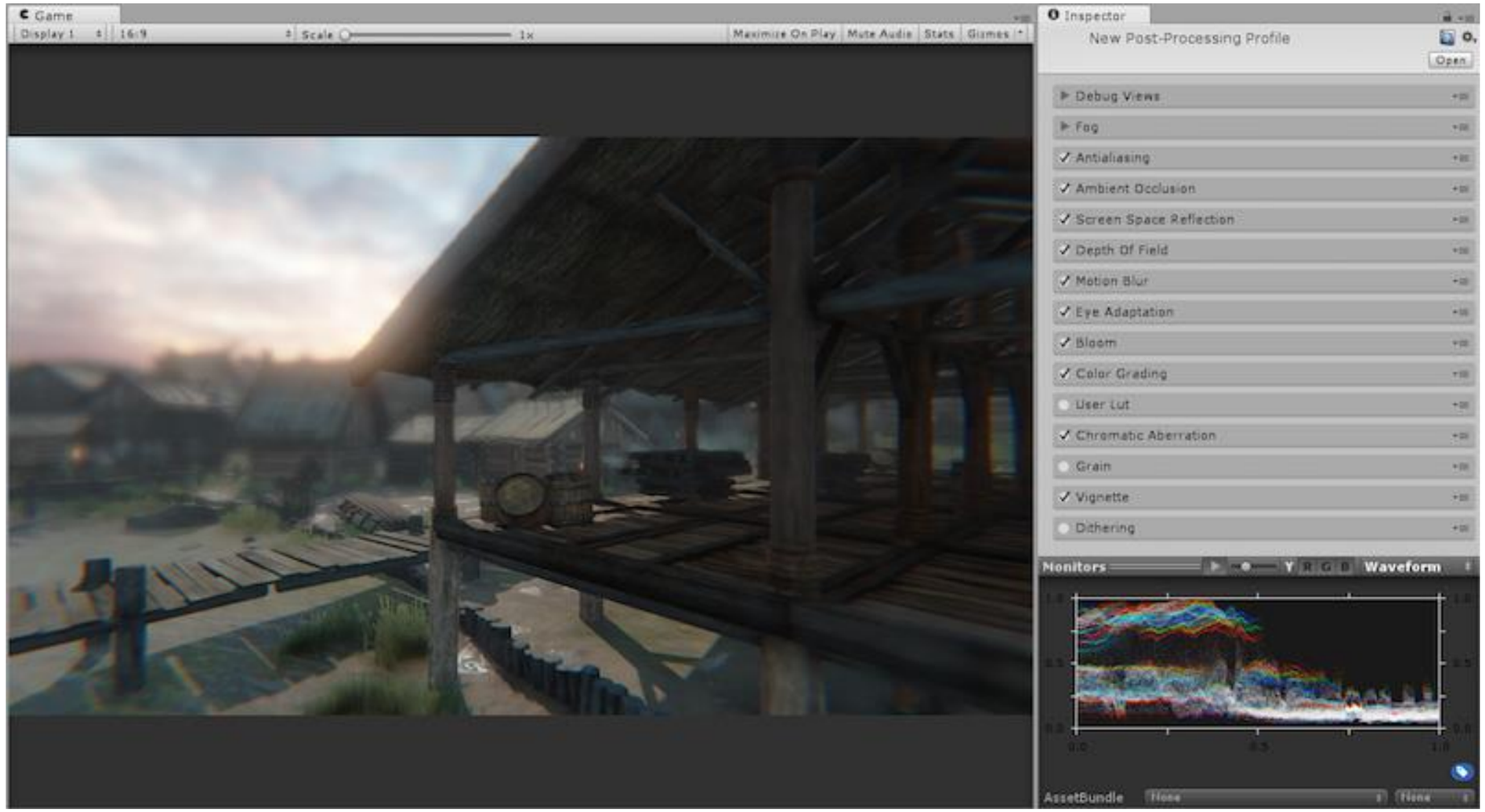
- ❑ Use a single skinned Mesh Renderer
- ❑ Use as few Materials as possible
- ❑ Use as few bones as possible
- ❑ Minimize the polygon count
- ❑ Keep forward and inverse kinematics separate

Lighting performance

Use Lightmapping to “bake” static lighting just once.

- ❑ It runs a lot faster (2–3 times faster for 2-per-pixel lights)
- ❑ It looks a lot better, as you can bake **global illumination** and the **lightmapper** can smooth the results

Do not use post processing stack

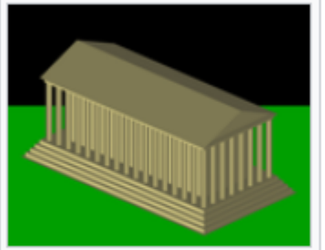


GPU: Texture compression and mipmaps

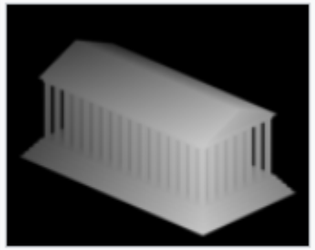
- Use smaller texture.
- Use Compressed textures to decrease the size of your textures.
  - This can result in faster load times, a smaller memory footprint, and ***dramatically increased rendering performance***.
  - Compressed textures only use a fraction of the memory bandwidth needed for uncompressed 32-bit RGBA textures.

Do not use real-time soft shadow

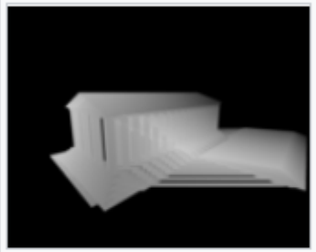
Realtime shadows are nice, but they can have a high impact on performance, both in terms of extra draw calls for the CPU and extra processing on the GPU.



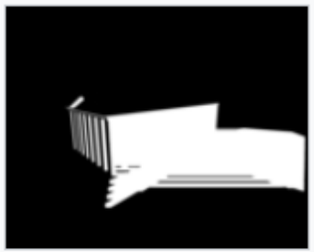
Scene rendered from the light view.



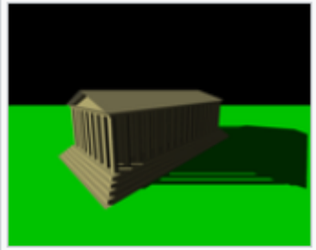
Scene from the light view, depth map.



Visualization of the depth map projected onto the scene



Depth map test failures.



Final scene, rendered with ambient shadows.

Checklist to make your game faster

- ❑ Keep the vertex count below 200K and 3M per frame when building for PC (depending on the target GPU).
- ❑ If you're using built-in shaders, pick ones from the Mobile or Unlit categories. They work on non-mobile platforms as well, but are simplified and approximated versions of the more complex shaders.
- ❑ Keep the number of different materials per scene low, and share as many materials between different objects as possible.
- ❑ Set the Static property on a non-moving object to allow internal optimizations like static batching
- ❑ Only have a single (preferably directional) pixel light affecting your geometry, rather than multiples.
- ❑ Bake lighting rather than using dynamic lighting.

Checklist to make your game faster

- ❑ Use compressed texture formats when possible, and use 16-bit textures over 32-bit textures.
- ❑ Avoid using fog where possible.
- ❑ Use Occlusion Culling to reduce the amount of visible geometry and draw-calls in cases of complex static scenes with lots of occlusion. Design your levels with occlusion culling in mind.
- ❑ Use skyboxes to “fake” distant geometry.
- ❑ Use pixel shaders or texture combiners to mix several textures instead of a multi-pass approach.
- ❑ Use half precision variables where possible.
- ❑ Minimize use of complex mathematical operations such as pow, sin and cos in pixel shaders.
- ❑ Use fewer textures per fragment.

Demo

MY **BRIGHT** FUTURE

DSU Dongseo University 동서대학교

DSU Dongseo University 동서대학교