

08 Optimizing and Debugging

Contents

[Overview 3](#_Toc508201619)

[8.1. General Profiling Tips 4](#_Toc508201620)

[8.2. Gameplay Debugging 5](#_Toc508201621)

[8.3. Blueprint Debugging 6](#_Toc508201622)

[8.4. Optimizations 6](#_Toc508201623)

[Exercises 8](#_Toc508201624)

[Exercise 8A: Profiling Performance 8](#_Toc508201625)

[Exercise 8B: Debugging Blueprints 13](#_Toc508201626)

[Multiple-Choice Quiz 14](#_Toc508201627)

[Optimization 14](#_Toc508201628)

[Debugging 15](#_Toc508201629)

# Overview

**PURPOSE**

The purpose of this tutorial is to outline features and functionality available to profile, debug, and optimize your projects within Unreal Engine.

**SCOPE**

This section will provide an introductory overview of the following:

* Profiling tools
* Debugging tools
* Optimization techniques

**PREREQUISITES**

* A basic understanding of the engine and how to use the tools
* Infiltrator demo, available in the Learn tab of the Launcher
* Showdown demo, available in the Learn tab of the Launcher

## 8.1. General Profiling Tips

Ideally, you should profile as close to the target you care about as possible. For example, a good profiling case is testing your game in Standalone form on the target hardware with built lightmaps.

For good profiling, it is best to set up a reproducible case isolated from things that can add noise or overhead to the results. Even the Editor adds some overhead (for example, an open Content Browser can add rendering cost), so it is better to profile in a game directly. In some cases, it might even be useful to change code (to disable random number generators, for instance). The Pause command or Slomo 0.001 can be very useful in making things more stable.

Measure a few times to know how accurate your profiling is. Stat commands such as stat unit and stat fps give you some numbers to start with. Accurate profiling should be done in milliseconds (ms), not frames per second (fps). You easily can convert between the numbers, but relative improvements have little meaning when measured in fps.

If you see a limit on 30 fps (~33.3ms) or 60 fps (~16.6ms), you likely have VSync enabled. For more accurate timing, it is better if you profile without it.

Do not expect an extremely high frame rate with a very simple scene. Many design choices pay off for complex scenes (deferred rendering, for example) but have a higher base cost. You also might bump into a frame rate limiter. If needed, such things can be adjusted (for instance, t.MaxFPS, r.VSync).

|  |  |
| --- | --- |
| C:\Users\Melanie Nikdel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tips-png-4.png | **Need More?**   * For tips and guidelines on setting up content and levels for performance, see <https://docs.unrealengine.com/latest/INT/Engine/Performance/Guidelines/index.html>. * For information on the stat commands, check out <https://docs.unrealengine.com/latest/INT/Engine/Performance/StatCommands/index.html>. |

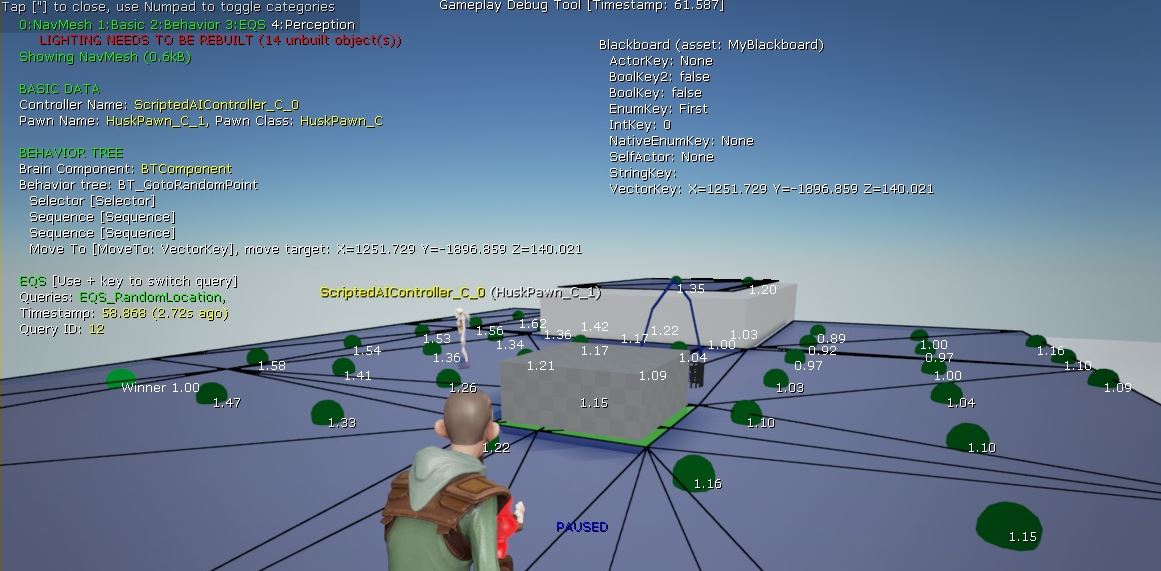
## 8.2. Gameplay Debugging

The Gameplay Debugger is useful for watching real-time data at runtime, even on clients in networked games using replication. It works in Play in Editor, Simulate in Editor, and standalone game sessions, and all of the data is displayed overlaid on the game viewport. The system provides a framework that can be extended to enable debugging of game-specific data.

Engine implementation can show the following:

* Basic data from Pawn
* Basic data from AIController
* Information about Behavior Tree and Blackboard data
* Information about executed Environment Queries (EQS)
* Information from perception system
* Navmesh around player or selected Pawn with all details like links or areas

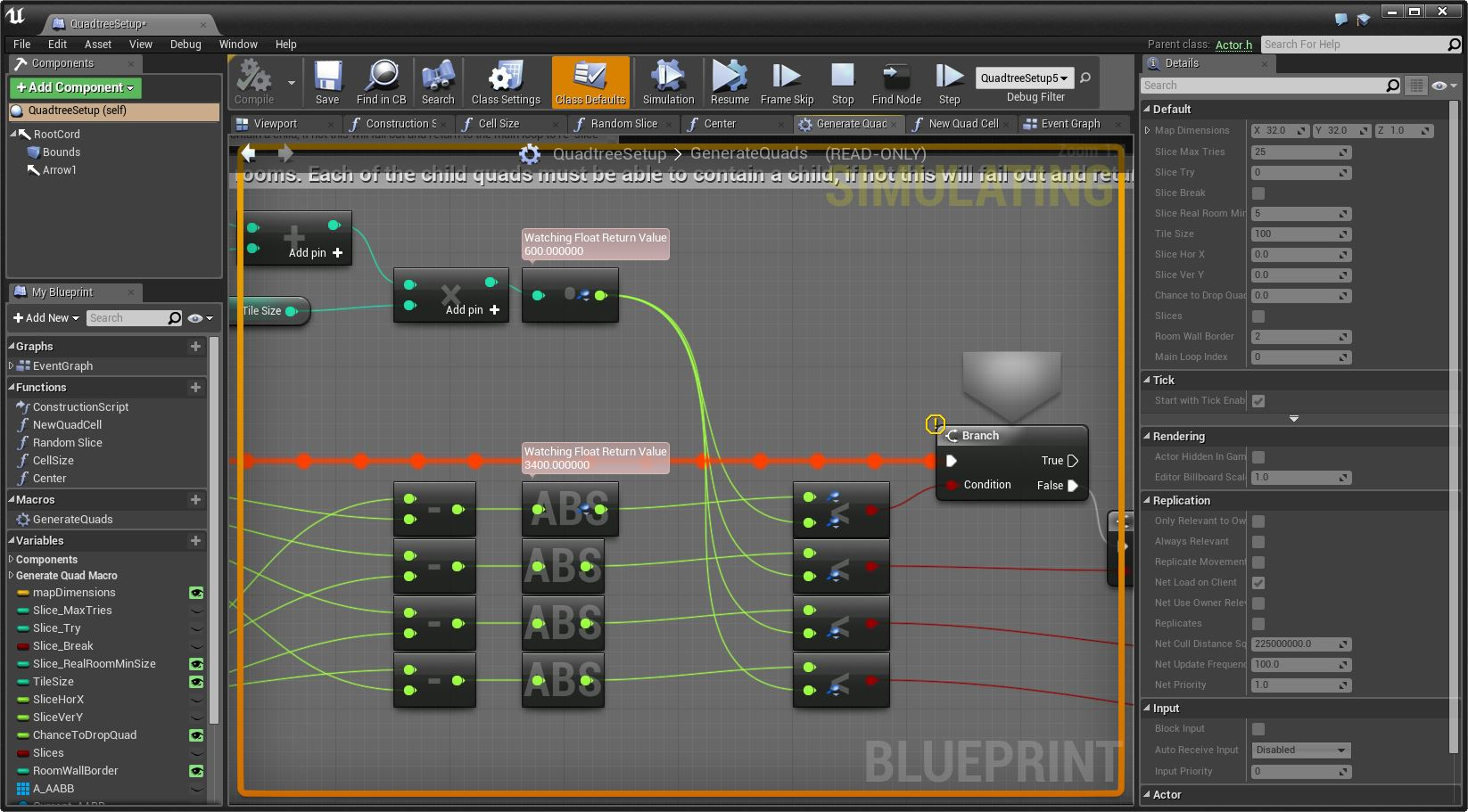
Usually there is a lot of data, so GDT uses categories to limit the amount of information on screen.



Screenshot captured on a client running the Gameplay Debugger

## 8.3. Blueprint Debugging

Blueprint debugging is a very powerful feature that allows you to pause the execution of the game in “**Play in Editor**” or “**Simulate in Editor**” mode and step through any graph of a Blueprint or Level Blueprint through the use of Breakpoints.



|  |  |
| --- | --- |
| C:\Users\Melanie Nikdel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tips-png-4.png | **Need More?**   * For more information on debugging with Blueprints, see   <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Debugging/index.html>.   * For a step-by-step example of Blueprint Debugging, check out   <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/BP_HowTo/Debugging/index.html>. |

## 8.4. Optimizations

Once you’ve identified an issue, it needs to be fixed. The fixes can vary drastically depending on the diagnosed issue; however, at a high level it’s common to ask yourself the following questions:

* Does this need to happen as frequently? Perhaps something can be cached and only intermittently updated. Perhaps something can be deemed irrelevant and not worked on until it’s strictly appropriate.
* Can this be done in conjunction with something else? Is it possible to batch-process it with something else, and, if it is, what are the outcomes of doing so?
* Does this need to be as accurate as it is? Would having an approximation produce a result that is less expensive and just as effective in a real-time environment?
* Is there a quick way to reduce the number of things I’m performing an action against? Is there a simple test to easily avoid performing more complicated ones?
* Can any information from the previous frame be used to aid the computation involved in this one?
* Can this be distributed over multiple frames? Do all 10 enemies need to spawn right now or could they be spread over a second or two?

Common Game Thread optimizations include the following:

* Reducing the number of ticking actors
* Reducing the tick rate for various actors
* Processing only actors that are relevant to the situation (for instance, not processing skeletal mesh animations of actors that are offscreen)
* Caching values to prevent unnecessarily rerunning functions
* Binding events to others so that a process is called only when necessary
* Distributing the workload of heavy functions across several frames
* Destroying components that will never be required again to reduce transform updates
* Pooling actors so that, for instance, instead of creating a new enemy, you are setting a previously beaten enemy from invisible to visible and setting their health back to full. This technique is often used for bullet Actors in games that have many projectiles.

Common rendering optimizations include the following:

* Batch rendering items that can be rendered in a single draw call through such processes as instancing or merging meshes
* Using simplified shader formats
* Using general assumptions in a shader to reduce instruction count for certain environments and then using a more advanced shader only when necessary. This optimization is common for certain lighting types or material types.
* Caching data to a texture for easy lookups instead of manually rerunning calculations that may be unnecessary
* Processing data per-vertex instead of per-pixel for objects that will be close to the player and occupying a lot of screen space

# Exercises

## Exercise 8A: Profiling Performance

**Deliverables:** Activity sheet (see below)

**Instructor Task:** Have the students profile the performance of both the Infiltrator and Showdown demos. Have them compare and contrast the performance of both demos by answering a selection of questions.

**Student Task:** Run and profile the performance of the Infiltrator and Showdown demos.



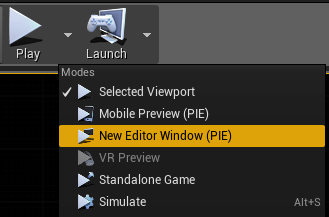
Showdown

Infiltrator

**Directions**

1. Download the Infiltrator and Showdown demos from the Marketplace tab in the launcher.
2. Create a new project for each, one at a time, and play through the demo (in the editor) to make sure everything works.
3. Run the playthrough again, this time using the CPU Profiler. This can be done by typing “**stat Game**” into the console.

**Note: Make sure you select “New Editor Window” as the Play mode.**



1. Make a note of the output show by the CPU profiler for each of the demos. You can capture screenshots and/or record your findings on the provided activity sheet.
2. Run the playthrough again, this time enabling the GPU Profiler. This can be done by pressing

“**Ctrl + Shift + Comma**” to bring up the profile window or, alternatively, by typing “**profilegpu**” into the console.

**Note: Make sure you select “New Editor Window” as the Play mode.**



1. Using your findings, answer the questions on the provided answer sheet.

Activity Sheet 8A

|  |  |  |  |
| --- | --- | --- | --- |
| Name: |  | Date: |  |

1. Screenshots (You can paste any relevant screenshots in the space below.)

|  |  |  |  |
| --- | --- | --- | --- |
| Name: |  | Date: |  |

1. Observations (Record any observations in the space below.)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |  |
| --- | --- | --- | --- |
| Name: |  | Date: |  |

1. Short-Answer Questions

Of the two demos you tested, which used the fewest hardware resources at runtime?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Which demo had the highest average GPU usage?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

According to your findings, what is the biggest performance bottleneck for each demo? Why?

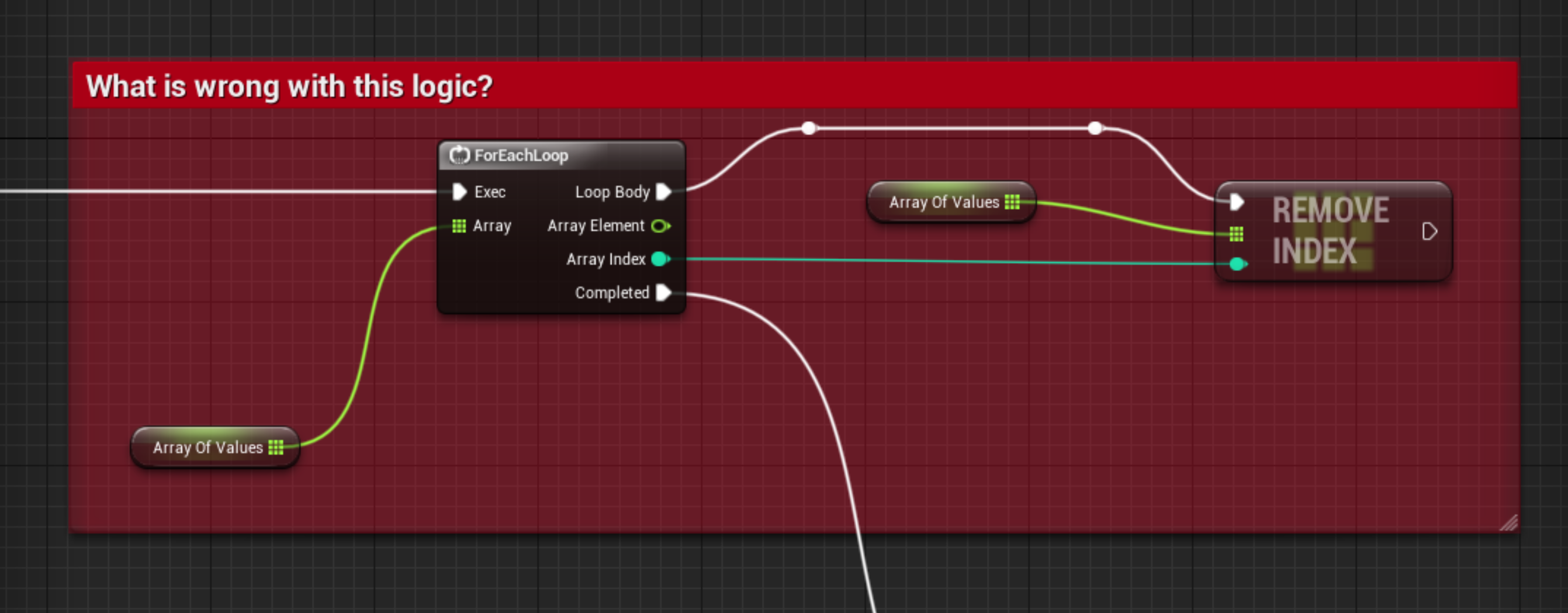
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Exercise 8B: Debugging Blueprints

**Deliverables:** Project files (uassets)

**Instructor Task:** Have the students debug the Blueprints in **Blueprints\DebuggingExamples\**

**Student Task:** Step through each of the issues and resolve their problems.



**Directions**

1. Open the Blueprint debugging examples and identify the issue in each
2. Add breakpoints to determine the issues in each of the assets
3. Implement a solution for each issue and save them for inspection.

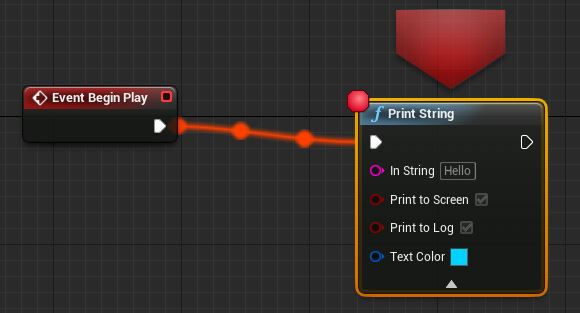
# Multiple-Choice Quiz

|  |  |  |  |
| --- | --- | --- | --- |
| Name: |  | Date: |  |

## Optimization

1. Changing rendered mesh complexity based on distance from the player view is also known as
   1. Level of Detail
   2. Occlusion
   3. Culling
   4. Tessellation
   5. None of the above
2. Stationary lights have the cheapest performance overhead per instance.
   1. True
   2. False
3. Which of the following best describes “Texture Packing”?
   1. Storing multiple textures as a single larger texture to reduce draw calls
   2. Storing additional information within the color channels of a texture
   3. Splitting a texture into smaller sections to more easily pack into memory
   4. Layering multiple low-resolution textures within a material to increase overall detail

## Debugging



1. What is the term given to the red dot in the above image?
   1. Breakpoint
   2. Pointbreak
   3. Debug Node
   4. Pointerbreak
2. Which of the following best describes a watch value?
   1. Watch values are a useful feature that allows you to time the execution of your code.
   2. A watch value references the current time against your blueprint time.
   3. Watch values allow you to view the real-time data stored in a variable.
   4. A watch value stores a copy of a selected variable to check the timing of the data.
3. Which of the following could interfere with the execution of your code?
   1. Unclear variable names
   2. Compiler error
   3. Logic error
   4. None of the above