Alan Rock

arock@SummitPeakTechnologies.com

Abstract

GPU Coder is a system for GPU development in Unity using C# with debugging support in Visual Studio. A programmer specifies desired fields, properties, and methods in a C# class, and GPU Coder will generate framework code for GPU development. Generated code supports a customizable UI Toolkit user interface and direct GPU library integration with object oriented design features. GPU Coder allows pure C# to run on the GPU and be debugged on the CPU without having to learn HLSL, GLSL, CUDA, ShaderLab, and all the intricacies of GPU programming.

GPU Coder

Rapid prototyping, development, and debugging system for compute shaders and vertex-fragment shaders for high-speed computation.

# Table of Contents

[Table of Contents 1](#_Toc105305484)

[Table of Figures 2](#_Toc105305485)

[UI Builder 3](#_Toc105305486)

[Horizontal vs Vertical Layout 3](#_Toc105305487)

[Organization 5](#_Toc105305488)

[G# Settings File 6](#_Toc105305489)

[Precompiled Libraries 6](#_Toc105305490)

[Tasks 6](#_Toc105305491)

[Auto-Generated Code 7](#_Toc105305492)

[Speed up compiling 7](#_Toc105305493)

[Language Enhancements 7](#_Toc105305494)

# Table of Figures

[Figure 1. A horizontal UI at the top works well for fewer UI controls. 3](#_Toc105305495)

[Figure 2. As the number of controls increase, the aspect ratio of the graphical display also increases. 4](#_Toc105305496)

[Figure 3. The number of UI controls can eventually cover the entire screen. 4](#_Toc105305497)

[Figure 4. A vertical UI limits the number of UI controls in the display and results in a graphical display with a lower aspect ratio. 5](#_Toc105305498)

[Figure 5. Unity allows the user to customize the UI display to some extent, but generally uses a vertical layout for the UI. 5](#_Toc105305499)

[Figure 6. The UXML file can be modified by G# so the UI can be displayed in the Game View and modified in UI Builder. 6](#_Toc105305500)

[Figure 7. Arrays can be specified as a class. 6](#_Toc105305501)

# UI Builder

## Horizontal vs Vertical Layout

G#9 will use UI Builder (UI Toolkit, UI Elements) for the user interface (UI).

Since the screen is narrower in the vertical direction, a vertical UI results in a larger graphical display area with a lower aspect ratio (more square appearance). A vertical UI limits the number of visible UI controls. This is visually less overwhelming but requires careful organization to avoid frequent scrolling and searching for UI controls.

A map of the world

Description automatically generated with medium confidence

Figure . A horizontal UI at the top works well for fewer UI controls.

Graphical user interface, application, table

Description automatically generated

Figure . As the number of controls increase, the aspect ratio of the graphical display also increases.

Graphical user interface, application, table, Excel

Description automatically generated

Figure . The number of UI controls can eventually cover the entire screen.

A white van on a road

Description automatically generated with medium confidence

Figure . A vertical UI limits the number of UI controls in the display and results in a graphical display with a lower aspect ratio.

Graphical user interface

Description automatically generated

Figure . Unity allows the user to customize the UI display to some extent, but generally uses a vertical layout for the UI.

## Organization

The UI should display in the Game View and in UI Builder. Building the UI from C# code at each compile will slow down compilation. G# should automatically rebuild the UI only when necessary.

This can be accomplished by modifying the UXML and USS files according to the G# settings file.

Text

Description automatically generated

Figure . The UXML file can be modified by G# so the UI can be displayed in the Game View and modified in UI Builder.

This allows the appearance of the UI to be customized in UI Builder.

It would be nice to allow the user to drag and dock UI containers at runtime, similar to the Unity editor.

# G# Settings File

It may be possible to avoid using a settings file. This would require generating all GPU compute and material shaders from the main code file. The settings file contents could be moved to a section of the main code file or placed in a separate partial class file.

When the settings file is changed, the changes can be automatically compiled. Much of the code is duplicated in the settings file and other files, but these duplications are automatically handed by G#.

## Precompiled Libraries

Library code, such as random number generators, signal processing, AI, or volumetric grid displays, can be directly imported and integrated into the application. Compile times can increase when many libraries are used. If a library changes, all the changes must be manually loaded to avoid breaking applications. Regex may simplify and speed up this process.

G#8 uses examples of precompiled libraries. Libraries are manually imported to try to speed up compilation. Compilation time was not significantly improved.

Text

Description automatically generated

Figure . Arrays can be specified as a class.

Arrays can be simplified as a class. UI.ShowIf statements can have errors as they are not compiled. Groups can be eliminated or used to group controls in foldouts.

# Tasks

## Auto-Generated Code

* Much of the code from OnValueChanged\_GS() can be auto-generated and moved to the \_.cs file. Allow this code to be added to the attributes in the \_GS file.
* Add pixel shader code from libraries automatically.
* Generate pixel shader code using multiple passes, so 3D objects such as spheres, capsules, cubes, etc can be shown
* Build UI by hand and use prefabs to generate UI. Try to find out why UI links get destroyed
* All UI code could be automatically generated for maximum flexibility
* Automatically generate show properties using text code in GS file. This is already done for displaying pixel shaders.

## Speed up compiling

* Only read a file once and write it once, and then only if necessary.
* It is possible to manipulate text on the GPU. I got it working from the editor, but it didn’t give much speedup
* Precompile library code.
  1. This can be done manually by making a partial class, split between multiple files, one file for each library.
     1. Add the library declaration in the settings file to generate the code
     2. Make a partial class for that library, copy the code from the child \_.cs class to the partial class.
     3. Remove the library declaration
* [https://github.com/cinight/MinimalCompute](https://github.com/cinight/MinimalCompute%20)
  + There are some interesting ways to speed up the GPU with AsyncGPUReadback, replace vertext buffer with StructuredBuffer
  + Use ComputeShader.SetConstantBuffer instead of gStruct to store all the compute shader variables. This will allow an extra buffer for kernels.
* <https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity5.html>
* StartsWith, EndsWith, and String.Format are extremely slow. Replace with custom versions to get 100X speedup.

## Language Enhancements

* Support inheritance for libraries. This is easier than adding a library declaration and then editing the generated code.
* Think about supporting recursion. Usually this can be replaced with a loop and a stack.
* Put support for buffers > 2Gb back into G#.
* Integrate into DOTS/ECS, use NetCode
* Support read/write race conflicts for a buffer. Automatically make a copy, update copy, read from copy, write to original.
* Require buffers to be passed to kernels: ref (read/write), out (write), otherwise (read)
  + The problem is that the method will be required to assign a value to out parameters.
  + However, a better way is to find all the buffers called from a kernel and from methods that the kernel calls.

Add classes and methods to G#. Instead of kernels and buffers, use a class to build a struct, then generate kernel functions from the methods.

struct dm\_Node { float3 position, velocity, position0, velocity0, force; uint u, matI; float pWaveVel, mass, stiff, fracture, strength, damp; }

The problem is that a nested class cannot access members of the parent class. This is needed to get the size. Perhaps make a partial class.

Is it possible to completely get rid of the \_GS settings file?

To rebuild the figures and table of contents, Ctrl-A, F9.

# Appendix

## UI Builder

To show in app

1. Project window, right click=>Create=>UI Toolkit=>Panel Settings Asset
2. Create GameObject, add UIDocument component, Assign Pannel Settings, Assign UXML file into Source Asset