

# Game Engine Architecture

## Chapter 10 Tools for Debugging and Development

# Logging and Tracing

- C/C++ programmers call this *printf debugging* (after the C standard library function, `printf()`)
- *printf debugging* is still a perfectly valid thing
- in real-time programming, it can be difficult to trace certain kinds of bugs using breakpoints and watch windows
- Some bugs are timing dependent:
  - they only happen when the program is running at full speed.
- Other bugs are caused by a complex sequence of events too long and intricate to trace manually one-by-one
  - the most powerful debugging tool is often a sequence of print statements

# teletype (TTY) output device

- Every game platform has some kind of console or teletype (TTY) output device.
  - In a console application written in C/C++, by printing to stdout or stderr via `printf()`, `fprintf()` or the C++ standard library's `iostream` interface.
  - `printf()` and `iostream` don't work if your game is built as a windowed application under Win32, because there's no console in which to display the output.
  - if you're running under the Visual Studio debugger, it provides a debug console to which you can print via the Win32 function `OutputDebugString()`.
  - On the PlayStation 3 and PlayStation 4, an application known as the Target Manager (or PlayStation Neighborhood on the PS4) runs on your PC and allows you to launch programs on the console.

# Formatted Output with OutputDebugString

```
int DebugPrintF(const char*
format, ...)
{
    va_list argList; //This type is
used as a parameter for the
macros defined in <cstdarg> to
retrieve the additional
arguments of a function.
    va_start(argList, format);
    //Initialize a variable argument
list
    int charsWritten =
VDebugPrintF(format, argList);
    va_end(argList); //End using
variable argument list
    return charsWritten;
}
```

```
int VDebugPrintF(const char*
format, va_list argList)
{
    const U32 MAX_CHARS = 1024;
    static char
s_buffer[MAX_CHARS];
    int charsWritten
= vsnprintf(s_buffer,
MAX_CHARS, format, argList);
    // Now that we have a
formatted string, call the
Win32 API.
    OutputDebugStringA(s_buffer);
    return charsWritten;
}
```

# Example

```
int main(int argc, char* argv[])
{
    int gameLevel = 1, gameSpeed = 60;
    OutputDebugString(L"Testing Formmated OutputDebugString... this will produce output in
the Output window of the VS Debugger");
    DebugPrintF("\nGameLevel: %d & GameSpeed: %d\n", gameLevel, gameSpeed);

    char buffer[50];
    int n, a = 5, b = 3;
    n = sprintf(buffer, "%d plus %d is %d", a, b, a + b);
    printf("[%s] is a string %d chars long. This is only good for Console applications and
not Win32!\n", buffer, n);

    char buf[4096];
    char* msgbuf = buf;
    sprintf(buf, "My gameLevel is %d\n", gameLevel);
    OutputDebugStringA(buf);

    return 0;
}
```

# Verbosity

- After adding a bunch of print statements to your code in strategically chosen locations, it's nice to be able to leave them there, in case they're needed again later.
- To permit this, most engines provide some kind of mechanism for controlling the level of *verbosity* via the command line, or dynamically at runtime.
- When the verbosity level is at its minimum value (usually zero), only critical error messages are printed. When the verbosity is higher, more of the print statements embedded in the code start to contribute to the output.

# Verbosity Demo

```
int g_verbosity = 0;
void VerboseDebugPrintF(int verbosity, const char* format, ...)
{
    // Only print when the global verbosity level is high enough.

    if (g_verbosity >= verbosity)
    {
        va_list argList; //This type is used as a parameter for the
        macros defined in <cstdarg> to retrieve the additional arguments of
        a function.
        va_start(argList, format); //Initialize a variable argument list
        VDebugPrintF(format, argList);
        va_end(argList); //End using variable argument list
    }
}
```

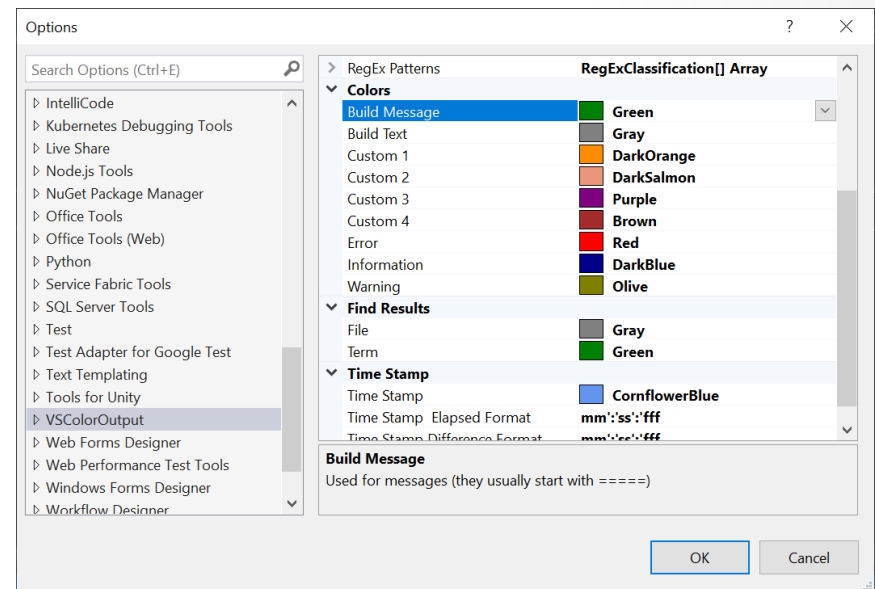
# Channels

- Categorize your debug output into *channels*.
- One channel might contain messages from the animation system, while another might be used to print messages from the physics system.
- Easy for a developer to focus in on only the messages he/she wants to see.
- PlayStation debug output can be directed to one of 14 distinct TTY windows.
- Windows provide only a single debug output console.
- The output from each channel might be assigned a different color.
- *Filters* can be turned on and off at runtime, and restrict output to only a specified channel or set of channels



# VSColorOutput

- Visual Studio 2019
  - <https://marketplace.visualstudio.com/items?itemName=MikeWard-AnnArbor.VSColorOutput>
- Visual Studio 2022
  - <https://marketplace.visualstudio.com/items?itemName=MikeWard-AnnArbor.VSColorOutput64>
- Colors are set in the *Tools*
  - *Options*
  - *VSColorOutput* dialog



# Using Redis to Manage TTY Channels

- <http://redis.io>
- Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.
- Supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams.

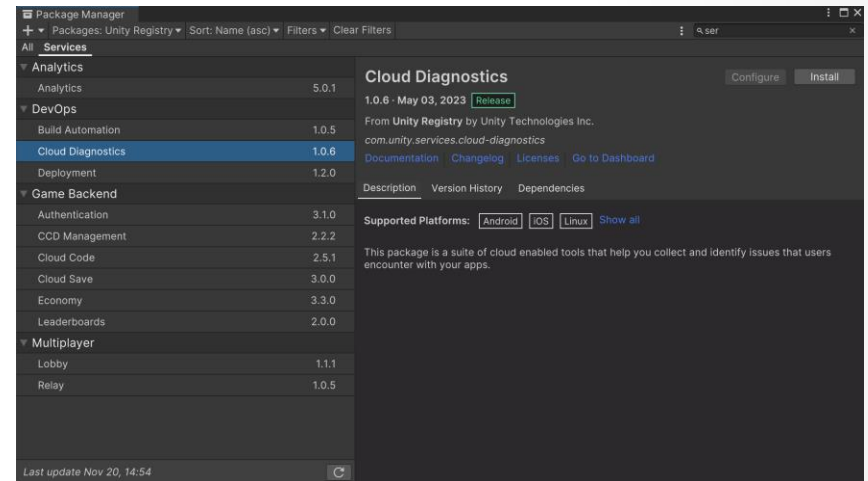
# Crash Reports

- Current level(s) being played at the time of the crash.
- World-space location of the player character when the crash occurred.
- Animation/action state of the player when the game crashed.
- Gameplay script(s) that were running at the time of the crash. (This can be especially helpful if the script is the cause of the crash!)
- Stack trace. Most operating systems provide a mechanism for walking the call stack (although they are nonstandard and highly platform specific).
- With such a facility, you can print out the symbolic names of all non-inline functions on the stack at the time the crash occurred.
- State of all memory allocators in the engine (amount of memory free, degree of fragmentation, etc.). This kind of data can be helpful when bugs are caused by low-memory conditions, for example.
- Any other information you think might be relevant when tracking down the cause of a crash.
- A screenshot of the game at the moment it crashed.



# Cloud Diagnostics

- Unity Cloud Diagnostics is a suite of cloud-enabled tools that help collect and identify possible issues in your Unity-built game. Collect crash and exception reports as well as user feedback so you can better diagnose issues and ensure a smooth gameplay experience.
- Cloud Diagnostics includes:
  - Crash and Exception Reporting: Automated exception and native crash reports along with the ability to add debugging symbol files.
  - User Reporting: Bug reports and user feedback directly from testers and the people using your app.

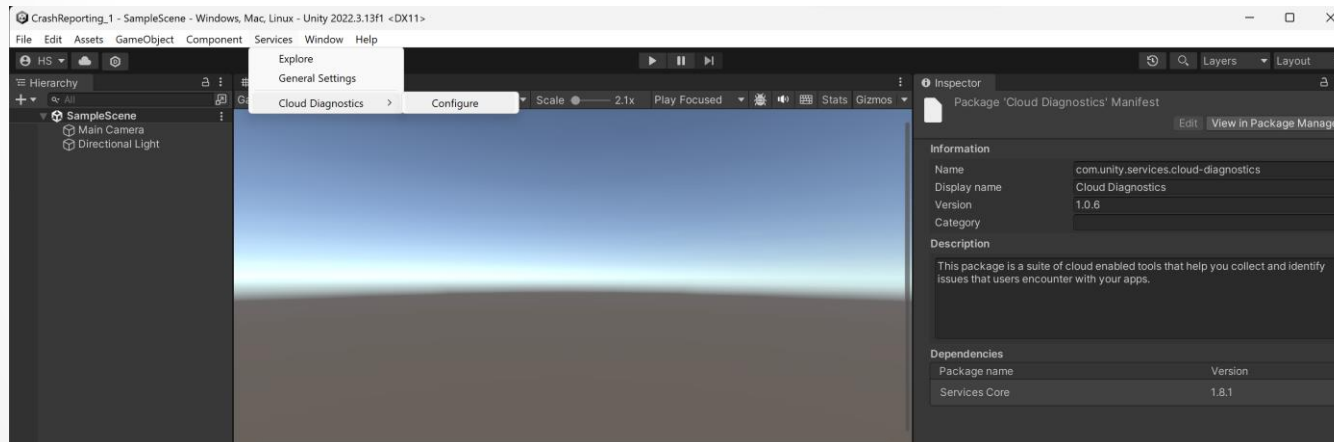


# Unity plans and Cloud Diagnostics tiers

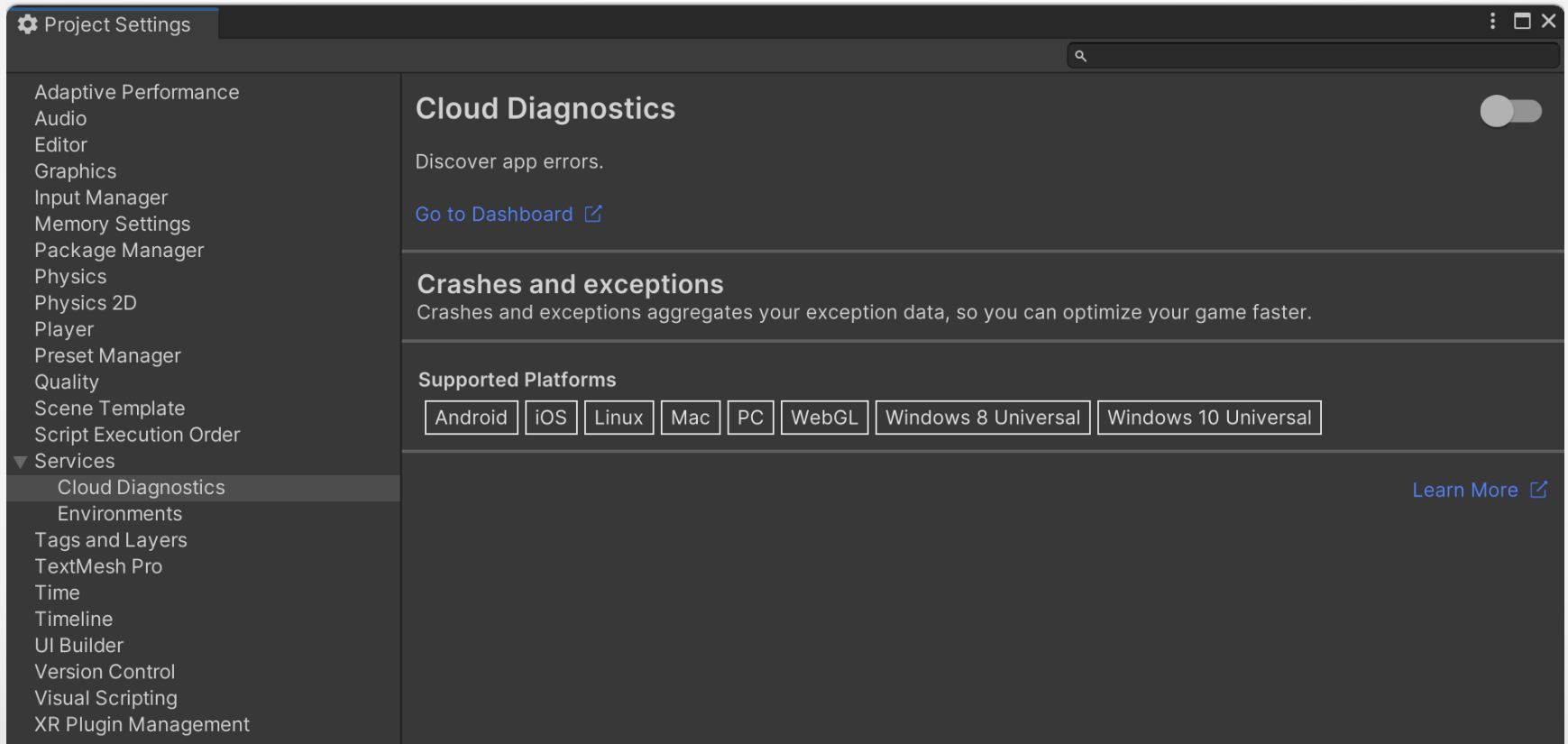
- Cloud Diagnostics is a service automatically bundled with your Unity plan, whether you use Unity Personal, Pro, or Plus. Note that while Cloud Diagnostics is available across all Unity plans, feature capacity differs across Unity subscription types. Here's a breakdown of what's included with Cloud Diagnostics depending on your Unity plan:
- <https://docs.unity.com/ugs/en-us/manual/cloud-diagnostics/manual/GettingStarted/GettingStartedwithCloudDiagnostics>

# Setting Up Crash and Exception Reporting In Unity

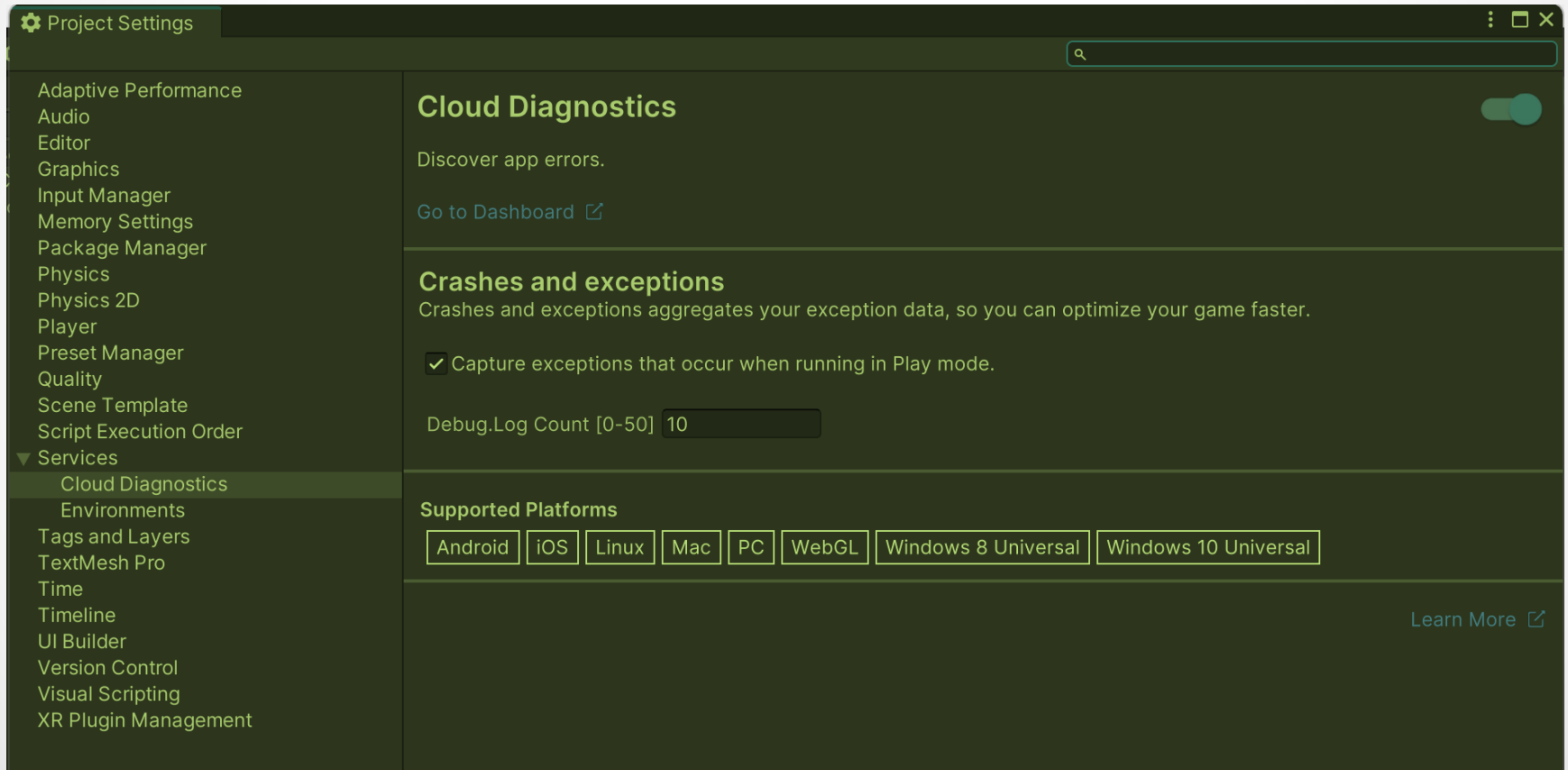
- <https://unity.com/products/unity-cloud>
- To enable Crash and Exception Reporting:
  - Install Cloud Diagnostics from the package manager
  - Open your project in the Unity Editor.
  - From the Unity Editor menu, select Services, Cloud Diagnostics, Configure.



# Cloud Dashboard



# Turn on the toggle in the project settings





# Crash and Exception Reporting

The screenshot displays the Unity Cloud Diagnostics interface for Crash and Exception Reporting. The browser address bar shows the URL: `cloud.unity.com/home/organizations/3573484136207/projects/ef037867-c799-4e36-80c6-06a504738aaf/cloud-diagnostics/crashes-excep...`. The left sidebar contains navigation links: Dashboard, Projects, Products, Administration, and Cloud Diagnostics (selected). The main content area is titled "Crash and Exception Reporting" and includes a "Add integration" button. It displays a progress bar for "Previous day reports" (0 of 10000 total, 0%) and a subscription status: "Subscription type: Plus/Pro/Enterprise" with details on report limits. Below this, four summary cards show "Total native crashes", "Users affected", "Total exceptions", and "Users affected", all with a value of 0. A filter section shows "Tag (1)" and "Clear" and "Apply" buttons. The "Occurrences" section includes a "Delete reports" button and a date range filter for "Nov 13 2023 - Nov 20 2023". The bottom of the interface shows the Unity logo and a "Documentation" link.

cloud.unity.com/home/organizations/3573484136207/projects/ef037867-c799-4e36-80c6-06a504738aaf/cloud-diagnostics/crashes-excep...

Cloud

Dashboard

Projects

Products

Administration

Shortcuts

Cloud Diagnostics

GAMING SERVICES

Cloud Diagnostics

User Reporting

Crash and Exception Reporting

Debugging Symbols

CrashReporting\_1

Crash and Exception Reporting

Add integration

Previous day reports

0 of 10000 total

0%

Subscription type: Plus/Pro/Enterprise

Cloud Diagnostics - Plus/Pro/Enterprise with Unity Pro or Plus includes up to 10000 reports per day.

Total native crashes

0

Users affected

0

Total exceptions

0

Users affected

0

Add filter

Tag (1)

Clear

Apply

Occurrences

Delete reports

Nov 13 2023 - Nov 20 2023

Documentation

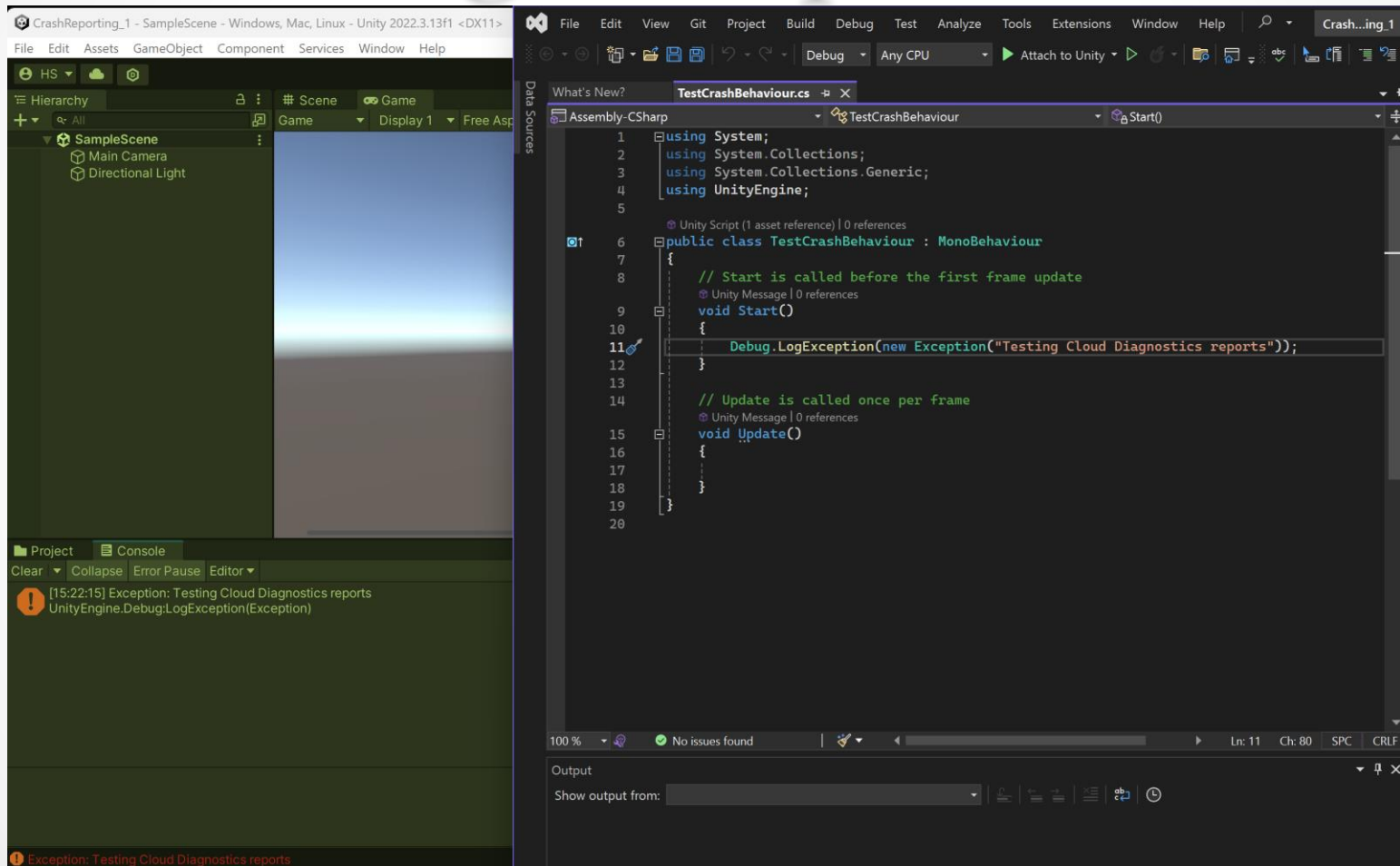
About

Unity

# Triggering a test report

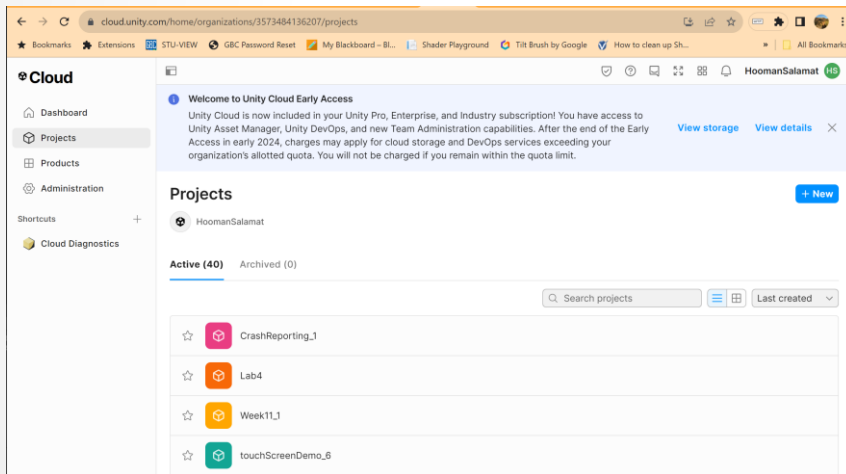
- Test the Cloud Diagnostics service by triggering a report and viewing it within the Unity Dashboard.
- Create a report by throwing an exception or logging an exception message via the `Debug.LogException()` method.
- To do this, locate a method within your C# script where you'd like the test exception to occur, such as the one that runs when the first screen is loaded, and add the following line:
  - `Debug.LogException(new Exception("Testing Cloud Diagnostics reports"));`
- Save the script and run your game in Play Mode. You should be able to view the `Debug.LogException` message in your console within the Editor.

# LogException



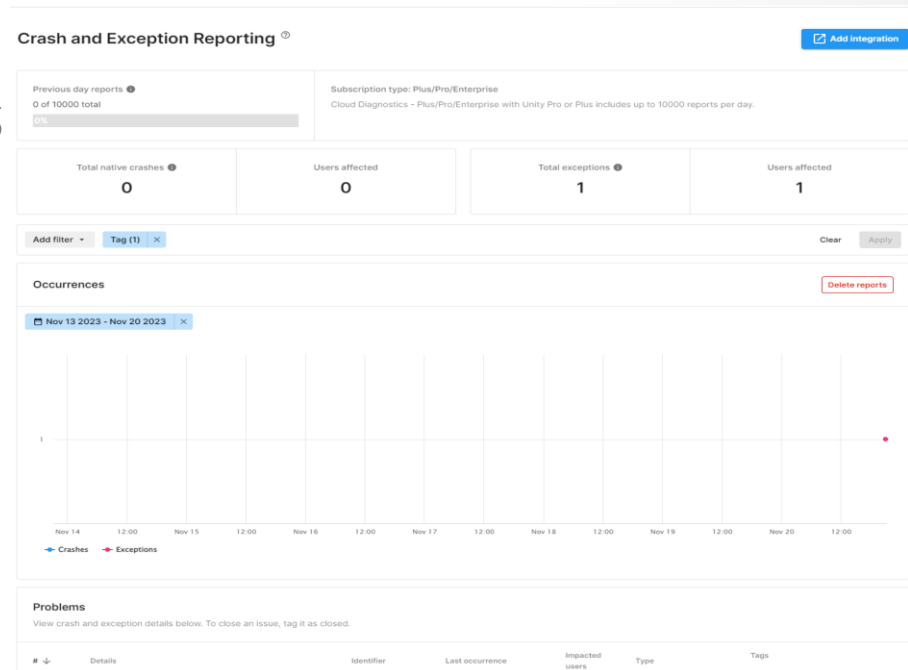
# Viewing your test report

- View Cloud Diagnostic reports from the Unity Dashboard:
- Open the Unity Dashboard.
- Select the project enabled with Cloud Diagnostics from the project selector on the top navigation menu.
- Your test report will be located in the Problems section. Select the title of the test report to view its data



# Problem Report

- Your test report will be located in the Problems section.
- Select the title of the test report to view its data.



## ← Problem Report

Identifier	Message	Type	Events	Impacted users	Last seen	Tags
15c33e32	Exception: Testing Cloud Diagnostics reports TestCrashBehaviour:Start() (at Assets/TestCrashBehaviour.cs:11)	Exception	1	1	3 minutes ago	<a href="#">+ Add tag</a>

Add filter Tag (1) x Clear Apply

# Setting up notifications for new reports

The screenshot displays the Unity Cloud web interface. The left sidebar shows the 'Cloud' navigation menu with options like Dashboard, Projects, Products, Administration (selected), Shortcuts, and Cloud Diagnostics. The main content area is titled 'Integrations' and shows a list of existing integrations. A 'NEW INTEGRATION' button is visible. A modal window titled 'New Integration' is open, showing a three-step process: 1. Select an integration, 2. Select an event, and 3. Configure options. Under 'Select an integration', there are radio button options for Webhook, Discord, Slack, Email, JIRA, Trello, and Microsoft Teams. The 'Webhook' option is selected. A 'NEXT' button is at the bottom right of the modal.

cloud.unity.com/home/organizations/3573484136207/projects/ef037867-c799-4e36-80c6-06a504738aaf/settings/integrations

Bookmarks Extensions STU-VIEW GBC Password Reset My Blackboard - BL... Shader Playground Tilt Brush by Google How to clean up Sh... All Bookmarks

Cloud

Administration

Dashboard

Projects

Products

Administration

Shortcuts

Cloud Diagnostics

Administration

Organization

Organization settings

Organization members

Service accounts

Service usage

Unity Cloud storage

Project integrations

Notification Settings

Finance and billing

Billing overview

Cost and usage reporting

Budget alerts

Payment methods

Transaction history

Fair usage reporting

CrashReporting\_1

Integrations

Project: CrashReporting\_1 UUID: ef037867-c799-4e36-80c6-06a504738aaf

NEW INTEGRATION

New Integration

1 Select an integration 2 Select an event 3 Configure options

Select the tool you'd like to integrate with

☒ Webhook  
A general-purpose mechanism that allows Unity to send POST requests to external services.

☐ Discord  
Send notifications to your team's Discord channel. Require channel's webhook permission.

☐ Slack  
Send notifications to your team's Slack channel.

☐ Email  
Send email notifications to your team. Let them know to expect email notifications.

☐ JIRA  
Automatically create, update, and comment on JIRA issues.

☐ Trello  
Automatically create, update, and comment on Trello cards.

☐ Microsoft Teams  
Send notifications to your team's Microsoft Teams channel.

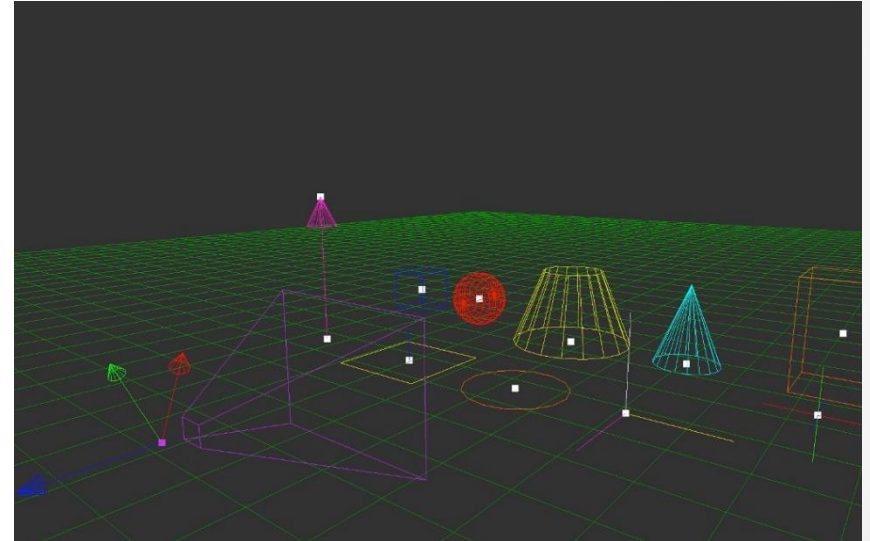
NEXT

# Debug Drawing Facilities

- Modern interactive games are driven almost entirely by math.
- Most good game engines provide an API for drawing colored lines, simple shapes and 3D text.
- We call this a *debug drawing* facility, because the lines, shapes and text that are drawn with it are intended for visualization during development and debugging
- With a debug drawing API, logical and mathematical errors become immediately obvious. “a picture is worth a thousand minutes of debugging”

# Debug Drawing API

- It should support a useful set of primitives
  - Lines,
  - Spheres,
  - *Points* (usually represented as small crosses or spheres, because a single pixel is very difficult to see),
  - Coordinate axes (typically, the x-axis is drawn in red, y in green and z in blue),
  - Bounding boxes, and
  - Formatted text.





# Debug Drawing API

- It should provide a good deal of flexibility in controlling how primitives are drawn, including:
  - color,
  - line width,
  - sphere radii,
  - the size of points, lengths of coordinate axes, and dimensions of
  - other “canned” primitives.
- It should be possible to draw primitives in world space or in screen space
- It should be possible to draw primitives with or without *depth testing* enabled.
- It should be possible to make calls to the drawing API from anywhere in your code.
- Every debug primitive should have a *lifetime* associated with it.
- Handling a large number of debug primitives efficiently. When you’re drawing debug information for 1,000 game objects, the number of primitives can really add up!

# Example

```
class DebugDrawManager
{
public:
// Adds a line segment to the debug drawing queue.
void AddLine(const Point& fromPosition,
const Point& toPosition,
Color color,
float lineWidth = 1.0f,
float duration = 0.0f,
bool depthEnabled = true);
// Adds an axis-aligned cross (3 lines converging at
// a point) to the debug drawing queue.
void AddCross(const Point& position,
Color color,
float size,
float duration = 0.0f,
bool depthEnabled = true);
// Adds a wireframe sphere to the debug drawing queue.
void AddSphere(const Point& centerPosition,
float radius,
Color color,
float duration = 0.0f,
bool depthEnabled = true);
// Adds a circle to the debug drawing queue.
void AddCircle(const Point& centerPosition,
const Vector& planeNormal,
float radius,
Color color,
float duration = 0.0f,
bool depthEnabled = true);
```

```
// Adds a set of coordinate axes depicting the
// position and orientation of the given
// transformation to the debug drawing queue.
void AddAxes(const Transform& xfm,
Color color,
float size,
float duration = 0.0f,
bool depthEnabled = true);
// Adds a wireframe triangle to the debug drawing
// queue.
void AddTriangle(const Point& vertex0,
const Point& vertex1,
const Point& vertex2,
Color color,
float lineWidth = 1.0f,
float duration = 0.0f,
bool depthEnabled = true);
// Adds an axis-aligned bounding box to the debug
// queue.
void AddAABB(const Point& minCoords,
const Point& maxCoords,
Color color,
float lineWidth = 1.0f,
float duration = 0.0f,
bool depthEnabled = true);
// Adds an oriented bounding box to the debug queue.
void AddOBB(const Mat44& centerTransform,
const Vector& scaleXYZ,
Color color,
float lineWidth = 1.0f,
float duration = 0.0f,
bool depthEnabled = true);
// Adds a text string to the debug drawing queue.
void AddString(const Point& pos,
const char* text,
Color color,
float duration = 0.0f,
bool depthEnabled = true);
};
```

# Example

This global debug drawing manager is configured for drawing in full 3D with a perspective projection.

```
extern DebugDrawManager  
g_debugDrawMgr;
```

This global debug drawing manager draws its primitives in 2D screen space. The (x,y) coordinates of a point specify a 2D location on-screen, and the z coordinate contains a special code that indicates whether the (x,y) coordinates are measured in absolute

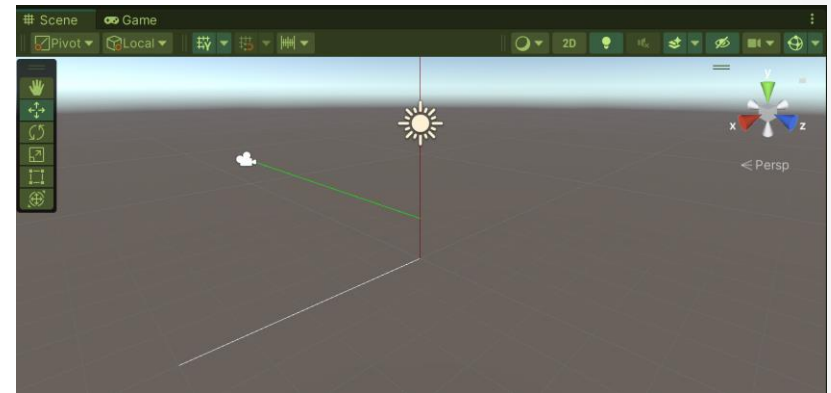
```
extern DebugDrawManager  
g_debugDrawMgr2D;
```

```
void Vehicle::Update()  
{  
    // Do some calculations...  
    // Debug-draw my velocity vector.  
    const Point& start = GetWorldSpacePosition();  
    Point end = start + GetVelocity();  
    g_debugDrawMgr.AddLine(start, end,  
        kColorRed);  
    // Do some other calculations...  
    // Debug-draw my name and number of  
    passengers.  
    {  
        char buffer[128];  
        sprintf(buffer, "Vehicle %s: %d passengers",  
            GetName(), GetNumPassengers());  
        const Point& pos = GetWorldSpacePosition();  
        g_debugDrawMgr.AddString(pos,  
            buffer, kColorWhite, 0.0f, false);  
    }  
}
```

# Unity Debug.DrawLine

- Draws a line between specified start and end points.
- The line will be drawn in the Game view of the editor when the game is running and the gizmo drawing is enabled.
- The line will also be drawn in the Scene when it is visible in the Game view.
- Leave the game running and showing the line. Switch to the Scene view and the line will be visible.
- Example: Draw a 5-unit white line from the origin for 2.5 seconds  
`Debug.DrawLine(Vector3.zero, new Vector3(5, 0, 0), Color.white, 2.5f);`

- `//Demo Draw Ray`
- `Vector3 forward =`  
`transform.TransformDirection(Vector3.forward) * 10;`
- `Debug.DrawRay(transform.position, forward,`  
`Color.green);`



# In-Game Menus

- Provide a system of *in-game menus that includes*:
  - toggling global Boolean settings,
  - adjusting global integer and floating-point values,
  - calling arbitrary functions, which can perform literally any task within the engine, and
  - bringing up submenus, allowing the menu system to be organized hierarchically for easy navigation.
- An in-game menu should be easy and convenient to bring up, perhaps via a simple button press on the joypad.
- Bringing up the menus usually pauses the game.

# In-Game Console

- provides a command-line interface to the game engine's features
- allowing a developer to view and manipulate global engine settings, as well as running arbitrary commands

# User interface (UI) in Unity

- Unity provides three UI systems that you can use to create user interfaces (UI) for the Unity Editor and applications made in the Unity Editor:
  - UI Toolkit
  - The Unity UI package (uGUI)
  - IMGUI

# UI Toolkit

- UI Toolkit is the newest UI system in Unity. It's designed to optimize performance across platforms, and is based on standard web technologies. You can use UI Toolkit to create extensions for the Unity Editor, and to create runtime UI for games and applications.
- UI Toolkit includes:
  - A retained-mode UI system that contains the core features and functionality required to create user interfaces.
  - UI Asset types inspired by standard web formats such as HTML, XML, and CSS. Use them to structure and style UI.
  - Tools and resources for learning to use UI Toolkit, and for creating and debugging your interfaces.
  - Unity intends for UI Toolkit to become the recommended UI system for new UI development projects, but it is still missing some features found in Unity UI (uGUI) and IMGUI.



# The Unity UI (uGUI) package

- The Unity User Interface (Unity UI) package (also called uGUI) is an older, GameObject-based UI system that you can use to develop runtime UI for games and applications.
- In Unity UI, you use components and the Game view to arrange, position, and style the user interface.
- It supports advanced rendering and text features.

# IMGUI

- Immediate Mode Graphical User Interface (IMGUI) is a code-driven UI Toolkit that uses the OnGUI function, and scripts that implement it, to draw and manage user interfaces.
- You can use IMGUI to create custom Inspectors for script components, extensions for the Unity Editor, and in-game debugging displays.
- It is not recommended for building runtime UI.
- Unity intends for UI Toolkit to become the recommended UI system for new UI development projects, but it is still missing some features found in Unity UI (uGUI) and IMGUI. These older systems are better in certain use cases, and are required to support legacy projects.

# UI system

- The core of UI Toolkit is a retained-mode UI system based on recognized web technologies. It supports stylesheets, and dynamic and contextual event handling.
- The UI system includes the following features:
  - Visual tree: An object graph, made of lightweight nodes, that holds all the elements in a window or panel. It defines every UI you build with the UI Toolkit.
  - Controls: A library of standard UI controls such as buttons, popups, list views, and color pickers. You can use them as is, customize them, or create your own controls.
  - Data binding system: A system links properties to the controls that modify their values.
  - Layout Engine: A layout system based on the CSS Flexbox model. It positions elements based on layout and styling properties.
  - Event System: A system communicates user interactions to elements, such as input, touch and pointer interactions, drag-and-drop operations, and other event types. The system includes a dispatcher, a handler, a synthesizer, and a library of event types.
  - UI Renderer: A rendering system built directly on top of Unity's graphics device layer.
  - Editor UI support: A set of components to create Editor UI.
  - Runtime UI Support: A set of components to create runtime UI.

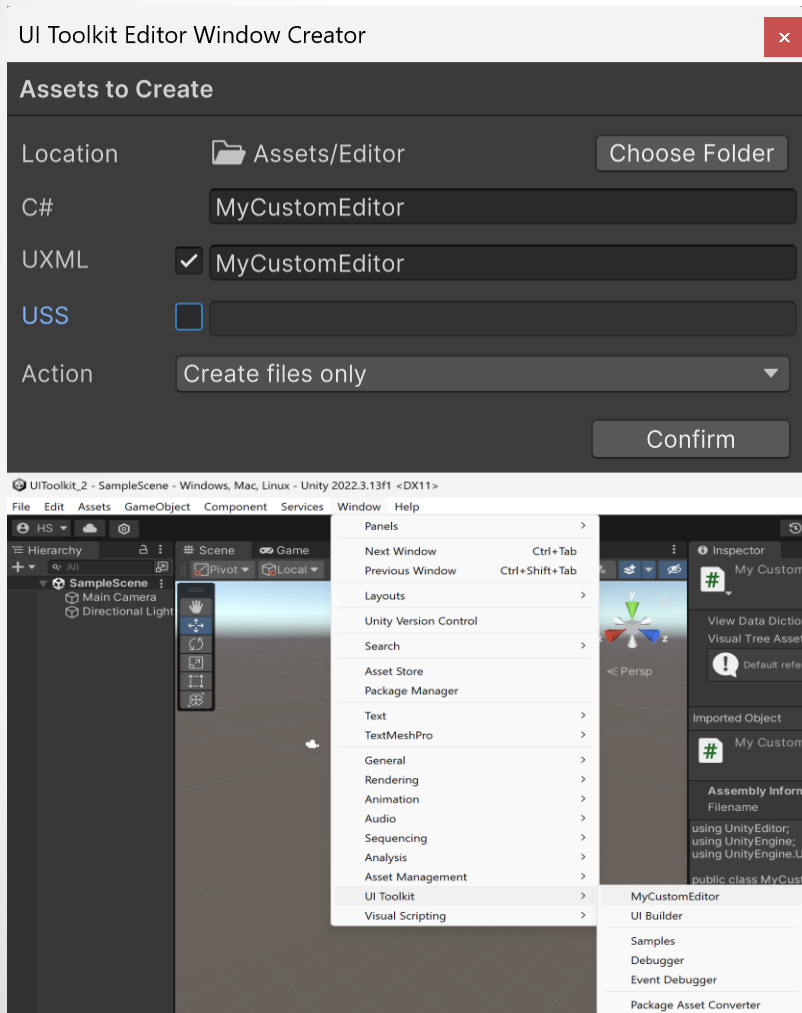
# UI assets

- Use the following asset types to build UI similar to how you develop web applications:
- UXML documents: HTML and XML inspired markup language defines the structure of UI and reusable UI templates. Although you can build interfaces directly in C# files, Unity recommends using UXML documents if possible.
- Unity Style Sheets (USS): Style sheets apply visual styles and behaviors to UI. They're similar to Cascading Style Sheets (CSS) used on the web, and support a subset of standard CSS properties. Although you can apply styles directly in C# files, Unity recommends using USS files if possible.

# UI tools and resources

- Use the following tools to create and debug your interfaces, and learn how to use the UI Toolkit:
- UI Debugger: A diagnostic tool that resembles a web browser's debugging view. Use it to explore a hierarchy of elements and get information about its underlying UXML structure and USS styles. You can find it in the Editor under Window > UI Toolkit > Debugger.
- UI Builder: A UI tool lets you visually create and edit UI Toolkit assets such as UXML and USS files.
- UI Samples: A library of code samples for UI controls that you can view in the Editor under Window > UI Toolkit > Samples.

# Create a custom Editor window



- Create a custom Editor window with two labels.
- Create a project in Unity Editor with any template.
- In the Project window, right-click in the Assets folder, and then select Create > UI Toolkit > Editor Window.
- In UI Toolkit Editor Window Creator, enter MyCustomEditor.
- Keep the UXML checkbox selected and clear the USS checkbox.
- Click Confirm.
- To open the Editor window, select Window > UI Toolkit > MyCustomEditor.

# Add UI controls to the window

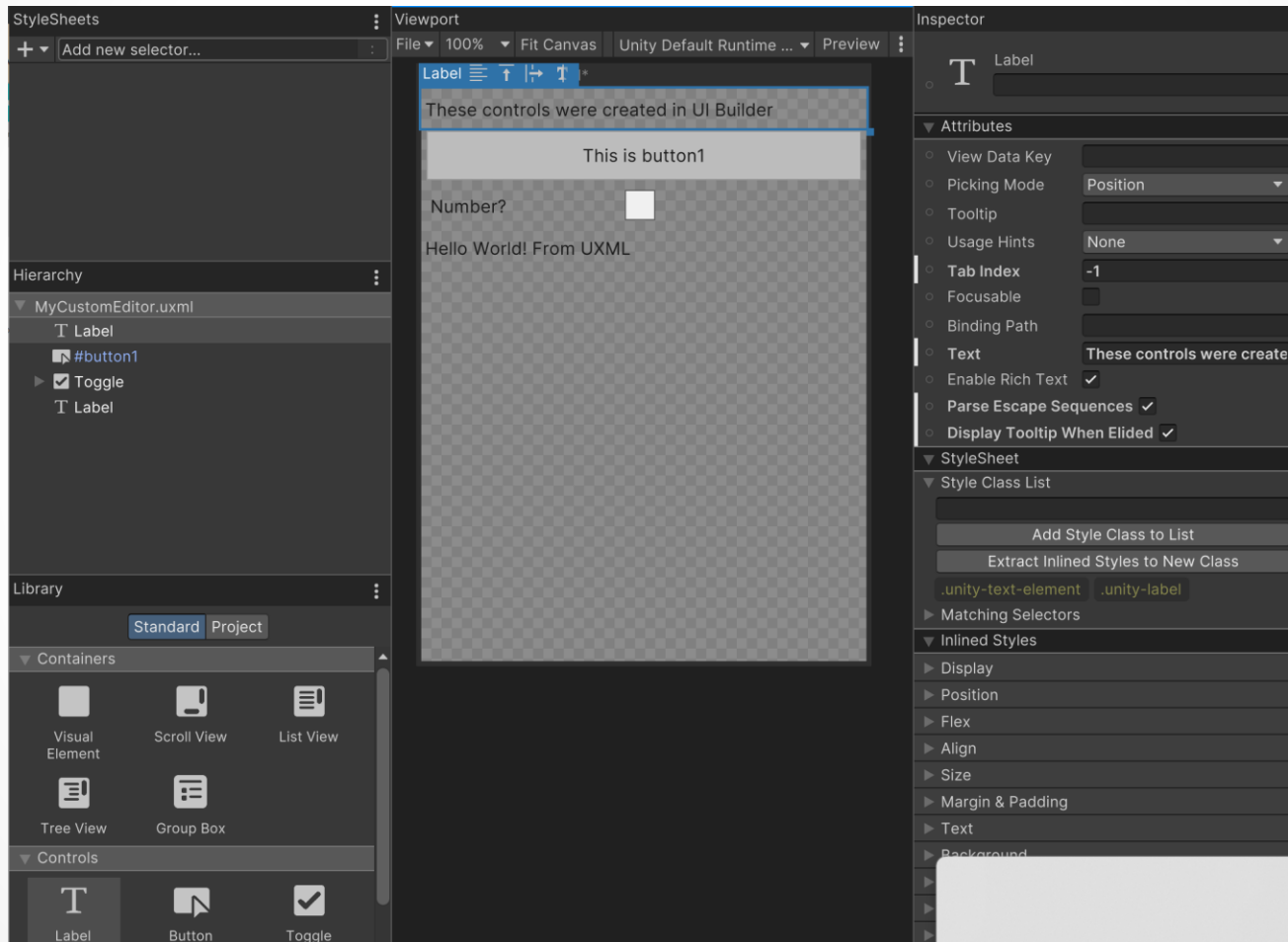
- You can add UI controls into your window in the following ways:
  1. Use the UI Builder to visually add the UI controls
  2. Use an XML-like text file (UXML) to add the UI controls
  3. Use C# script to add the UI controls

# Use UI Builder to add UI controls

- To visually add UI controls to your window, use UI Builder. The following steps add a button and a toggle into your custom Editor window in addition to the default label.
  1. In the Editor folder, double-click MyCustomEditor.xml to open the UI Builder.
  2. In the UI Builder, drag Button and Toggle from Library > Controls into the Hierarchy or the window preview in the Viewport.
  3. In the Hierarchy window, select Label.
  4. In the Inspector window, change the default text to These controls were created in UI Builder in the Text field.
  5. In the Hierarchy window, select Button.
  6. In the Inspector window, enter This is button1 in the Text field.
  7. Enter button1 in the Name field.
  8. In the Hierarchy window, select Toggle.
  9. In the Inspector window, enter Number? in the Label field.
  10. Enter toggle1 in the Name field.
  11. Save and close the UI Builder window.
  12. Close your custom Editor window if you haven't done so.
  13. Select Window > UI Toolkit > MyCustomEditor to re-open your custom Editor window to see the button and the toggle you just added.



# Custom Editor Window with one set UI Controls



# Use UXML to add UI controls

- If you prefer to define your UI in a text file, you can edit the UXML to add the UI controls. The following steps add another set of label, button, and toggle into your window.
- In the Editor folder, click Assets > Create > UI Toolkit > UI Document to create a UXML file called MyCustomEditor\_UXML.xml.
- Click the arrow on MyCustomEditor\_UXML.xml in the Project window
- Double-click inlineStyle to open MyCustomEditor\_UXML.xml in a text editor.
- Replace the contents of MyCustomEditor\_UXML.xml with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<engine:UXML
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xmlns:engine="UnityEngine.UIElements"
  xmlns:editor="UnityEditor.UIElements"
  xsi:noNamespaceSchemaLocation=" ../..UIElements
Schema/UIElements.xsd"
>
  <engine:Label text="These controls were created
with UXML." />
  <engine:Button text="This is button2"
name="button2"/>
  <engine:Toggle label="Number?"
name="toggle2"/>
</engine:UXML>
```

- Open MyCustomEditor.cs.
- Add a private VisualTreeAsset field called m\_UXMLTree to the MyCustomEditor class. Put the attribute [SerializeField] above it:

```
[SerializeField]
private VisualTreeAsset m_UXMLTree;
```

- Add the following code to the end of CreateGUI().  

```
root.Add(m_UXMLTree.Instantiate());
```
- In the Project window, select MyCustomEditor.cs.
- Drag MyCustomEditor\_UXML.xml from the Project window into the UXML Tree field in the Inspector. This assigns your UXML to the visual tree.
- Select Window > UI Toolkit > MyCustomEditor. This opens your custom Editor window with three labels, two buttons, and two toggles.

# Use C# script to add UI controls

- If you prefer coding, you can add UI Controls to your window with a C# script. The following steps add another set of label, button, and toggle into your window.
- Open MyCustomEditor.cs.
- Unity uses UnityEngine.UIElements for basic UI controls like label, button, and toggle. To work with UI controls, you must add the following declaration if it's not already present.
- using UnityEngine.UIElements;
- Change the text of the existing label from "Hello World! From C#" to "These controls were created using C# code.".
- The EditorWindow class has a property called rootVisualElement. To add the UI controls to your window, first instantiate the element class with some attributes, and then use the Add methods of the rootVisualElement.
- Your finished CreateGUI() method should look like the following:

```
public void CreateGUI()
{
    // Each editor window contains a root VisualElement
    object
    VisualElement root = rootVisualElement;

    // VisualElements objects can contain other
    VisualElements following a tree hierarchy.
    Label label = new Label("These controls were created
    using C# code.");
    root.Add(label);

    Button button = new Button();
    button.name = "button3";
    button.text = "This is button3.";
    root.Add(button);

    Toggle toggle = new Toggle();
    toggle.name = "toggle3";
    toggle.label = "Number?";
    root.Add(toggle);

    // Import UXML
    var visualTree =
    AssetDatabase.LoadAssetAtPath<VisualTreeAsset>("Assets/Ed
    itor/MyCustomEditor.uxml");
    VisualElement labelFromUXML =
    visualTree.Instantiate();
    root.Add(labelFromUXML);

    root.Add(m_UXMLTree.Instantiate());
}
```

# Debug Cameras and Pausing the Game

- An in-game menu or console system is best accompanied by two other crucial features:
  - the ability to detach the camera from the player character and fly it around the game world in order to scrutinize any aspect of the scene, and
  - The ability to pause, un-pause and single-step the game
- When the game is paused, it is still important to be able to control the camera.
  - we can simply keep the rendering engine and camera controls running, even when the game's logical clock is paused
- Slow motion mode is another incredibly useful feature for scrutinizing animations, particle effects, physics and collision behaviors, AI behaviors
  - put the game into slo-mo by simply updating the gameplay clock at a rate that is slower than usual.

# Cheats

- When developing or debugging a game, it's important to allow the user to break the rules of the game in the name of expediency.
- Such features are named *cheats*.
  - “pick up” the player character and fly him/her around in the game world, with collisions disabled.
  - *Invincible player*. As a developer, you often don't want to be bothered having to defend yourself from enemy characters, or worrying about falling from too high a height, as you test out a feature or track down a bug.
  - *Give player weapon*. It's often useful to be able to give the player any weapon in the game for testing purposes.
  - *Infinite ammo*. When you're trying to kill bad guys to test out the weapon system or AI hit reactions, you don't want to be scrounging for clips!
  - *Select player mesh*. If the player character has more than one “costume,” it can be useful to be able to select any of them for testing purposes.

# Screenshots and Movie Capture

- the ability to capture screenshots and write them to disk in a suitable image format such as Windows Bitmap files (.bmp), JPEG (.jpg) or Targa (.tga).
- make a call to the graphics API that allows the contents of the frame buffer to be transferred from video RAM to main RAM, where it can be scanned and converted into the image file format of your choice
- provide your users with various options controlling how screenshots are to be captured:
  - Whether or not to include debug primitives and text in the screenshot.
  - Whether or not to include heads-up display (HUD) elements in the screenshot.
  - The resolution at which to capture. Some engines allow high-resolution screenshots to be captured, perhaps by modifying the projection matrix so that separate screenshots can be taken of the four quadrants of the screen at normal resolution and then combined into the final high-res image.
  - Simple camera animations. For example, you could allow the user to mark the starting and ending positions and orientations of the camera.
  - A sequence of screenshots could then be taken while gradually interpolating the camera from the starting location to the ending location.

# In-Game Profiling

- Permits the programmer to annotate blocks of code that should be timed and give them human-readable names.
- A heads-up display is provided, which shows up-to-date execution times for each code block
- The display often provides the data in various forms, including raw numbers of cycles, execution times in microseconds, and percentages relative to the execution time of the entire frame.
- Many game engines provide an in-game profiling tool of some sort.
- The profiler measures the execution time of each annotated block via the CPU's hi-res timer and stores the results in memory.

# Hierarchical Profiling

- Computer programs are inherently *hierarchical*
- A function calls other functions, which in turn call still more functions.
- In C/C++, the root function is usually `main()` or `WinMain()`, although technically this function is called by a start-up function that is part of the standard C runtime library (CRT), so that function is the true root of the hierarchy.

```
void a()
```

```
{
```

```
  b();
```

```
  c();
```

```
}
```

```
void b()
```

```
{
```

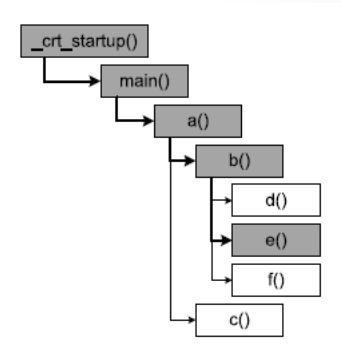
```
  d();
```

```
  e();
```

```
  f();
```

```
}
```

```
void c() { ... }
```





# Measuring Execution Times Hierarchically

- Execution time of a single function, the time we measure includes the execution time of any the child functions called and all of their grandchildren, and so on
- Measure both the *inclusive* and *exclusive* execution times of every function
- inclusive times measure the execution time of the function including all of its children, while exclusive times measure only the time spent in the function itself
- some profilers record how many times each function is called.

# Example

```
while (!quitGame)
{
    PollJoypad();
    UpdateGameObjects();
    UpdateAllAnimations();
    PostProcessJoints();
    DetectCollisions();
    RunPhysics();
    GenerateFinalAnimationPoses(
);
    UpdateCameras();
    RenderScene();
    UpdateAudio();
}
```

```
while (!quitGame)
{
    {
        PROFILE(SID("Poll Joypad"));
        PollJoypad();
    }
    {
        PROFILE(SID("Game Object Update"));
        UpdateGameObjects();
    }
    {
        PROFILE(SID("Animation"));
        UpdateAllAnimations();
    }
    {
        PROFILE(SID("Joint Post-Processing"));
        PostProcessJoints();
    }
    {
        PROFILE(SID("Collision"));
        DetectCollisions();
    }
    {
        PROFILE(SID("Physics"));
        RunPhysics();
    }
    {
        PROFILE(SID("Animation Finaling"));
        GenerateFinalAnimationPoses();
    }
    {
        PROFILE(SID("Cameras"));
        UpdateCameras();
    }
    {
        PROFILE(SID("Rendering"));
        RenderScene();
    }
    {
        PROFILE(SID("Audio"));
        UpdateAudio();
    }
}
```

# PROFILE() macro

```
struct AutoProfile
{
    AutoProfile(const char* name)
    {
        m_name = name;
        //The performance timer measures time in units called
        counts.
        QueryPerformanceCounter(&m_startTime);
    }

    ~AutoProfile()
    {
        LARGE_INTEGER endTime, DifferenceOfTime;
        QueryPerformanceCounter(&endTime);

        //QuadPart represents a 64-bit signed integer (for
        example: 670644208300)
        //Every LargeInteger is a union of High Part (long:156
        ) and Low Part (unsigned long: 629310124)

        DifferenceOfTime.QuadPart = endTime.QuadPart -
        m_startTime.QuadPart;
        cout << m_name << "Difference Of Time: " <<
        DifferenceOfTime.QuadPart << endl;
        //g_profileManager.storeSample(m_name, elapsedTime);
    }
    const char* m_name;
    LARGE_INTEGER m_startTime;
};

#define PROFILE(name) AutoProfile p(name)
```

- it breaks down when used within deeper levels of function call nesting
- if we embed additional PROFILE() annotations within the RenderScene() function, we need to understand the function call hierarchy in order to properly interpret those measurements.
- Solution: allow the programmer who is annotating the code to indicate the hierarchical interrelationships between profiling samples.

# In-Game Memory Stats and Leak Detection

- Game engines are also constrained by the amount of memory available on the target hardware
- Most game engines implement custom memory-tracking tools.
- Simply wrap `malloc()/free()` or `new/delete` in a pair of functions or macros that keep track of the amount of memory that is allocated and freed. Not that simple because:
  - *You often can't control the allocation behavior of other people's code*
  - *Memory comes in different flavors.*
  - *Allocators come in different flavors*
- If a model fails to load, a bright red text string could be displayed in 3D hovering in the game world where that object would have been.
- If a texture fails to load, the object could be drawn with an ugly pink texture that is very obviously not part of the final game.
- If an animation fails to load, the character could assume a special (possibly humorous) pose that indicates a missing animation, and the name of the missing asset could hover over the character's head.