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

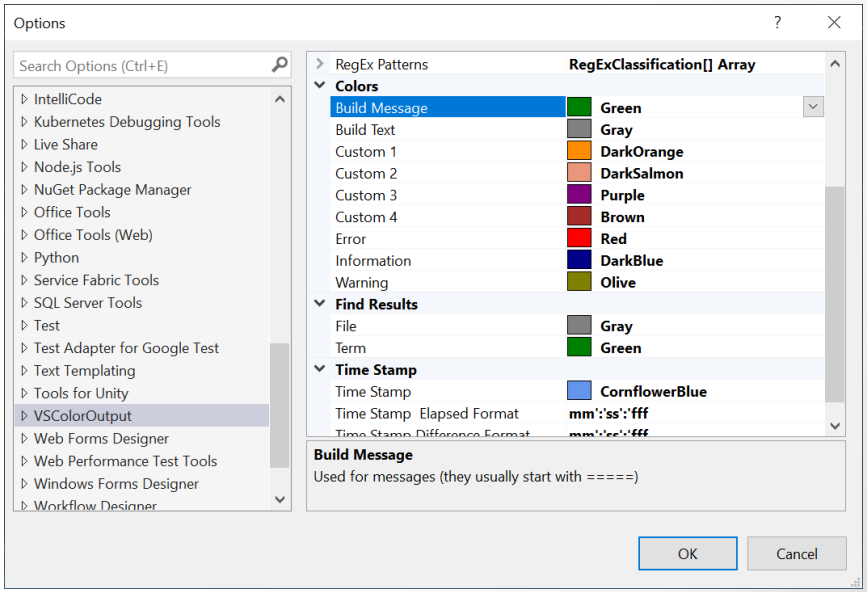
```
int g_verbosity = 0;
void VerboseDebugPrintF(int verbosity, const char* format, ...)
{
// Only print when the global verbosity level is high enough.

 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

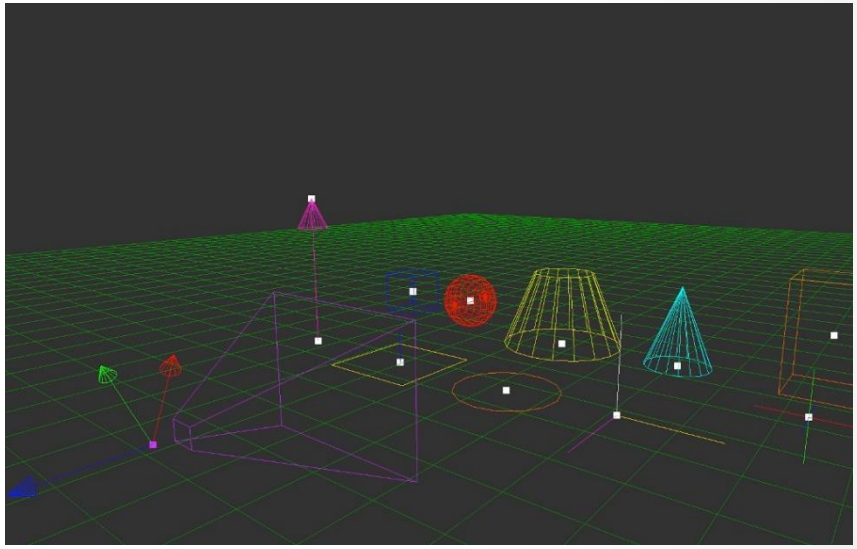
- Current level(s) being played at the time of the crash.
- World-space location of the player character when the crash occurred.
- Animation/action state of the player when the game crashed.
- Gameplay script(s) that were running at the time of the crash. (This can be especially helpful if the script 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.

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);
}
}
```

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

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.

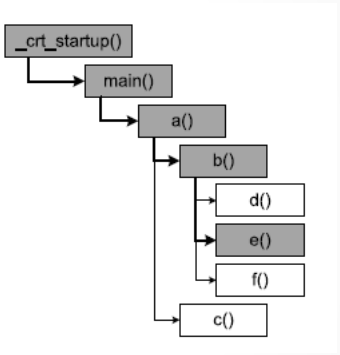
In-Game Profiling

- 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

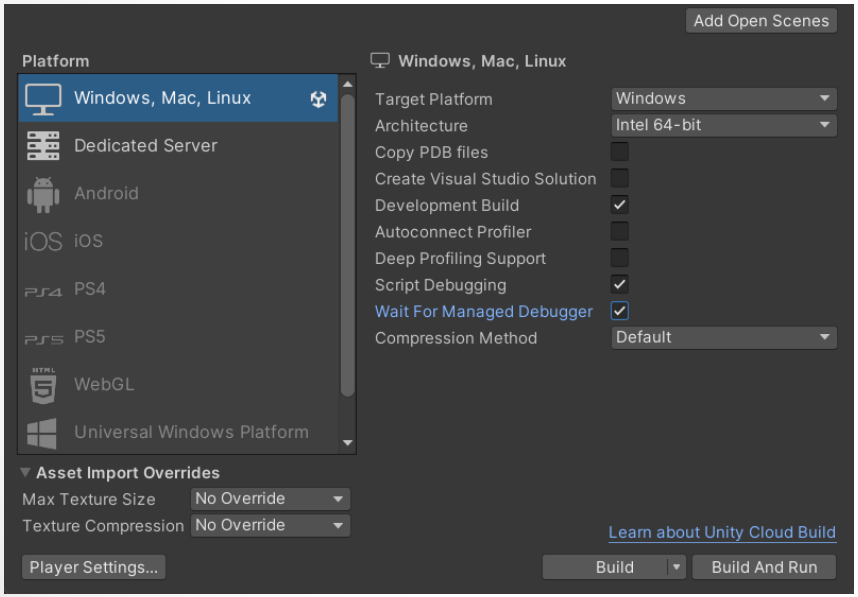
- Game engines are also constrained by the amount of memory available on the target hardware
- Most game engines implement custom memory-tracking tools.
- Simply wrap malloc()/free() or new/delete in a pair of functions or macros that keep track of the amount of memory that is allocated and freed. Not 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.

Debug C# code in Unity

- You can use a debugger to inspect your source code while your application is running. Unity supports the following code editors to debug C# code:

- Visual Studio (with the Visual Studio Tools for Unity plug-in)
- Visual Studio for Mac
- Jetbrains Rider
- Visual Studio Code
- Although these code editors vary slightly in the debugging features they support, they all provide basic functionality such as break points, single stepping, and variable inspection.

Debug in the Unity Player



- To compile a Unity Player for you to debug:

- Go to File > Build Settings.
- Enable the Development Build
- and Script Debugging options before you build the Player. You could also enable the Wait For Managed Debugger option to make the Player wait for a debugger to be attached before the Player executes any script code.
- Select Build And Run.

Unity Profiler

- The Unity Profiler is a tool you can use to get performance information about your application.
  - Profile your application in a player on your target platform
  - Profile your application in Play mode in the Unity Editor
  - Profile the Unity Editor

Profile your application on a target platform

- To profile your application on its target release platform, connect the target device to your network or directly to your computer via cable.
- You can also connect to a device via IP address.
- You can only profile your application as a Development Build.
- To set this up, go to Build Settings (menu: File > Build Settings) and select your application's target platform. Enable the Development Build setting.

The Profiler window

- To access the Unity Profiler, go to Window > Analysis > Profiler or use the keyboard shortcut Ctrl+7 (Command+7 on macOS).
- By default, the Profiler records and keeps the last 300 frames of your game, and shows you detailed information about every frame.
  - You can increase the number of frames it records in the Preferences window (menu: Edit > Preferences), up to 2,000 frames.

Asset Loading Profiler module

- The Asset Loading Profiler module displays information about how your application loads assets, including a breakdown of read operations by area.
- This module is not enabled by default. To enable the Asset Loading Profiler module, open the Profiler window, select the Profiler Modules dropdown menu, and toggle the Asset Loading checkbox.