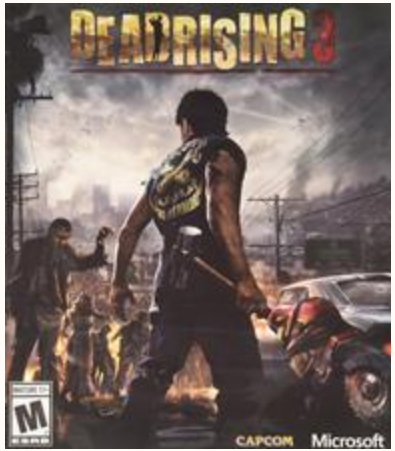
Building a 3D Game and Engine with D

Lewis Nicolle

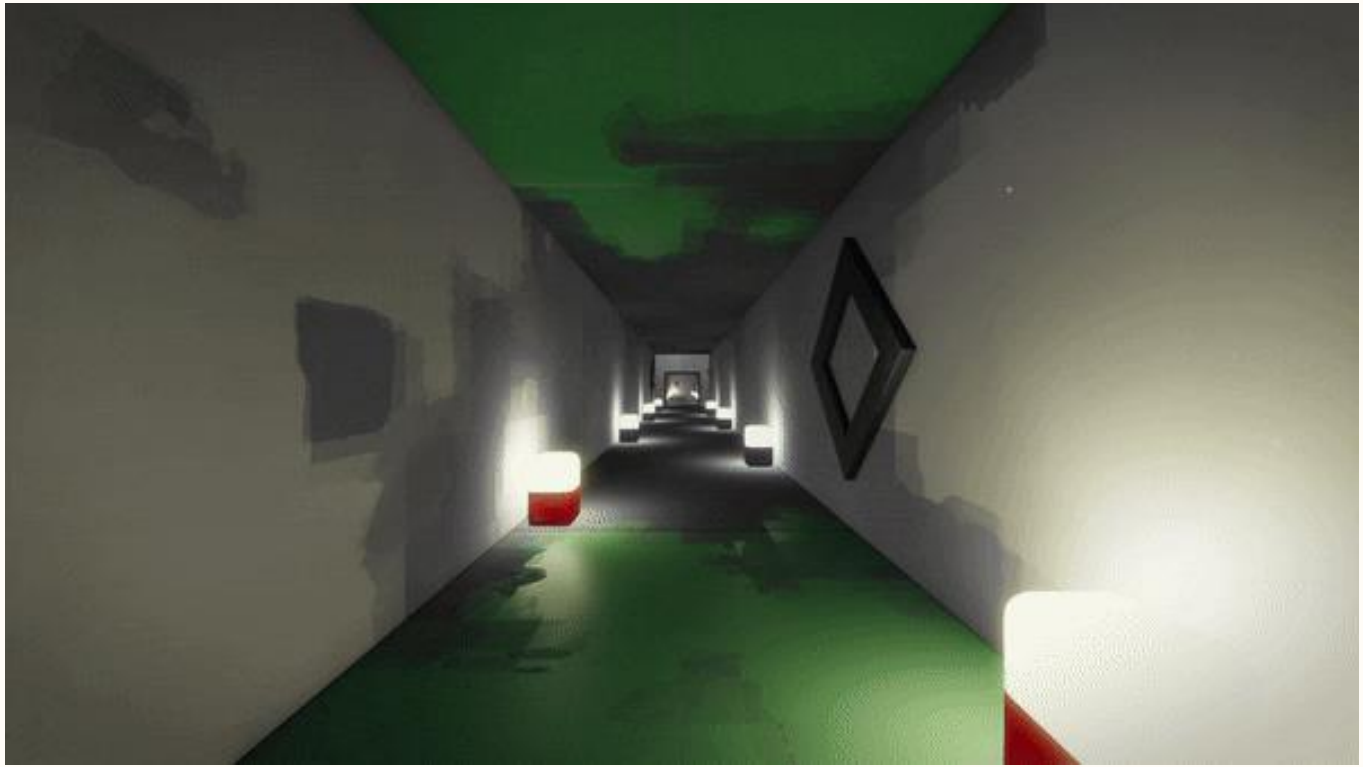
Hi, I'm Lewis

- Video game programmer specializing in rendering
- Worked in AAA at Capcom Vancouver, EA Vancouver, and Relic Entertainment



The Art of Reflection

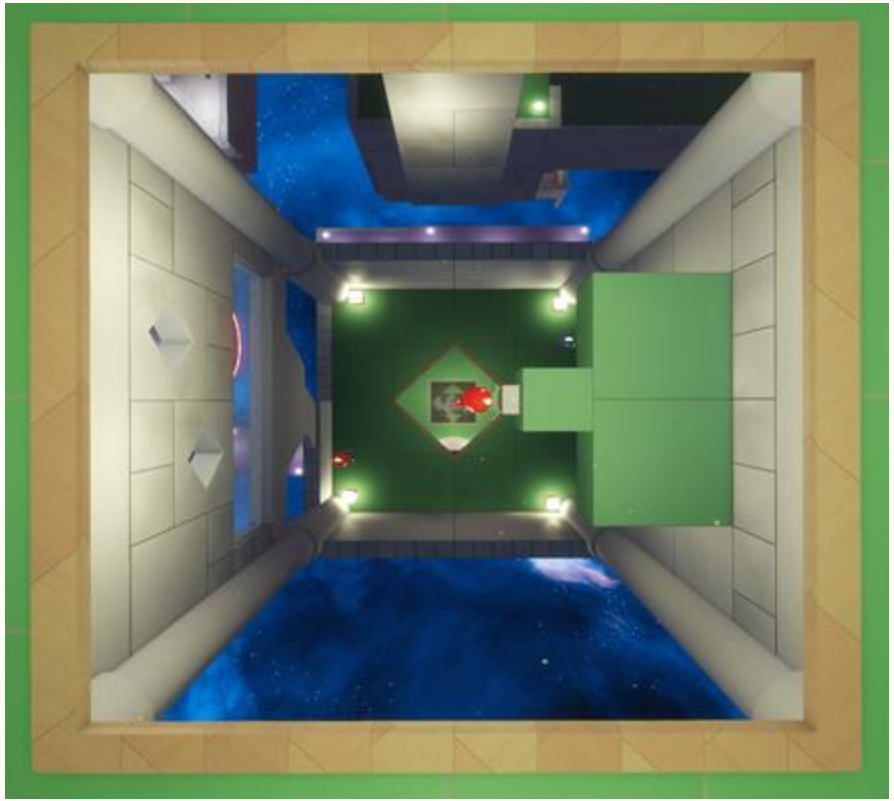
Live demo



https://store.steampowered.com/app/2290770/The_Art_of_Reflection/

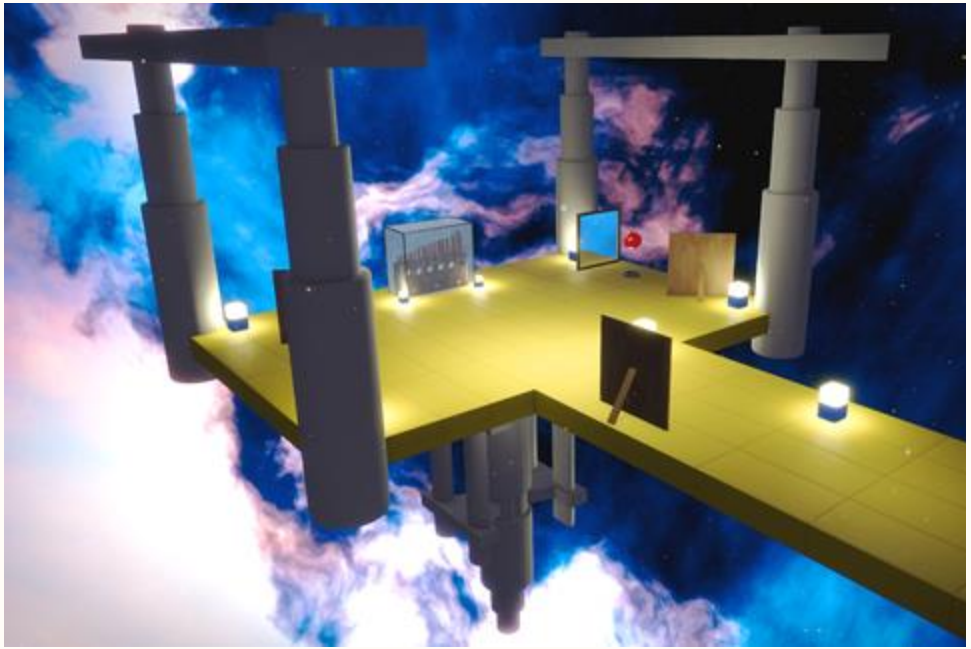
Technical goals

- Seamlessly teleport through mirrors
- Support 100+ mirror reflections on-screen at once
- Consistent 60fps minimum
- Reasonable support for older machines



Process goals

- Iteration time is king
- Do the simplest possible thing that works
- Preserve my ability to dive deep and customize or fix issues
- This is a marathon,
  I need tooling I'll be happy working with!



Why I chose D

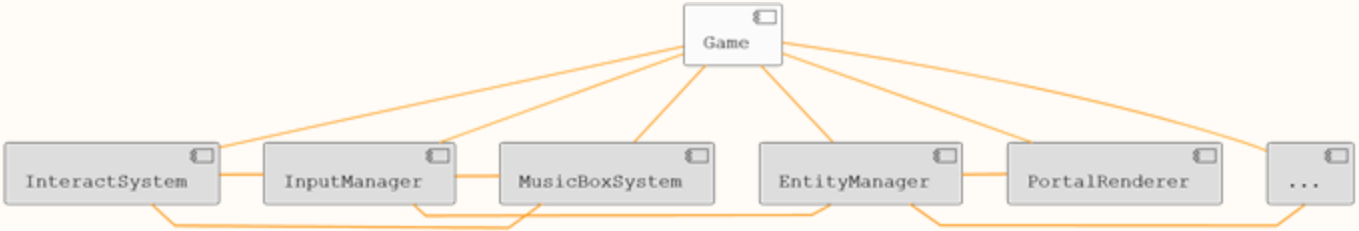
- Frustrations with C++
- Modern features (introspection, CTFE, slices, ranges, built-in bounds checking, imports, etc)
- High-level abstractions, while letting you drop down to the low level where needed
- Better standard library
- **Blisteringly fast compile times**
- **Mature language**
- **Flexible. Easy to trust D will let me do things the way I need to.**
- **Interop with C and C++**

How I use D

- **Keep it simple**
- Plain functions and structs
- Almost no exception handling, classes, inheritance hierarchies
- Use templates and CTFE, but judiciously

Basic engine structure

- The game is a collection of global systems
- Each system:
  - Gets a struct in the GameState and the Scene, with data allocated however the system wants
  - Opts in to game-wide functions (`Update(), Draw(), HandleEvent()`)
  - Presents a public interface
  - Freely makes global calls into other subsystems

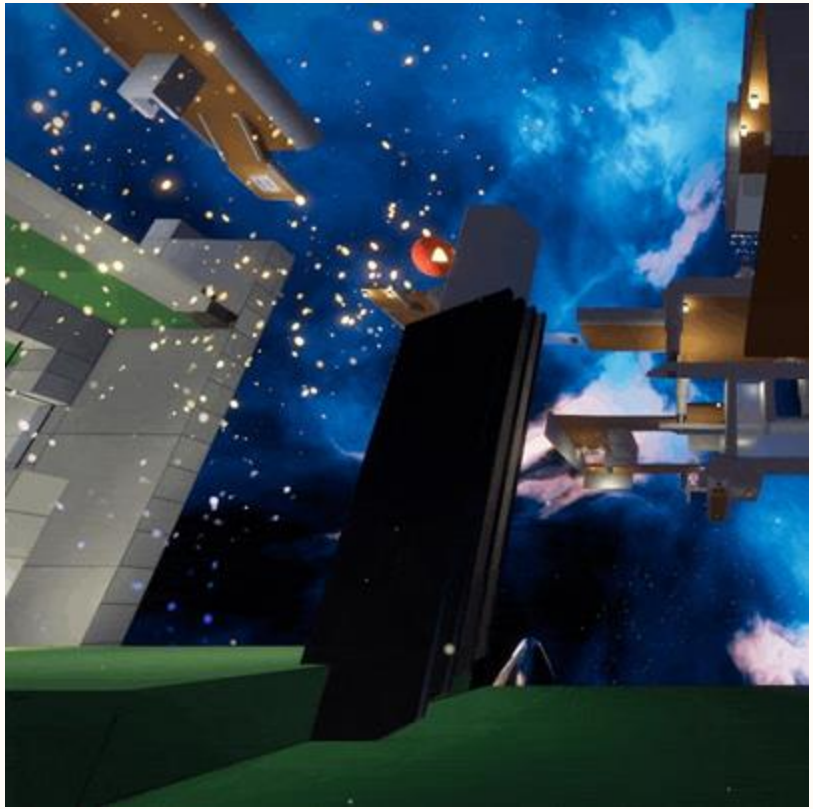


Memory management

- GC acts as global/permanent memory
- GC churn only for startup/debug/tools, never for regular gameplay
- `@nogc` is too restrictive, and doesn't know about my linear allocator
- Added debug printing when GC allocation occurs

Fun features: GameState inspector

- ~300 lines of CTFE/traits code
- New additions to gamestate automatically captured
- Confluence of features:
  - D introspection
  - Imgui
  - Code hotswapping



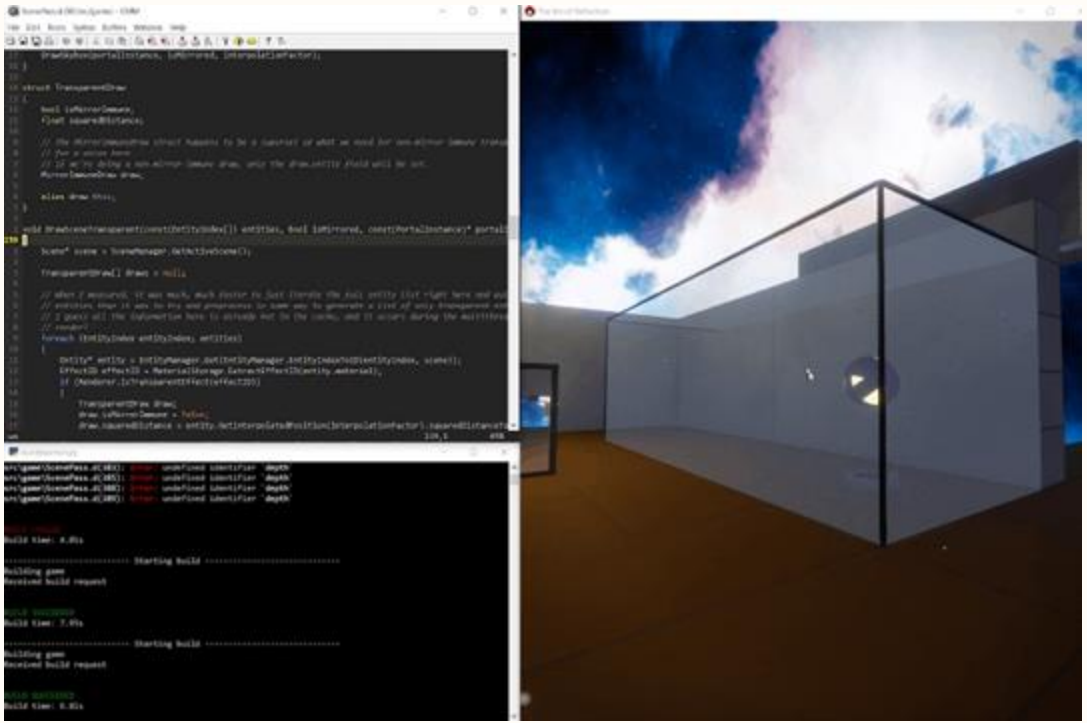
Fun features: Scrapheap

- Linear temporary allocator
- One per thread
- Cleared after each frame (or at end of worker task)
- Be careful nothing allocated in your scope expects to outlive the frame!
- Use an allocator stack to track the active allocator

```
string[] ProcessString(string input)
{
    mixin(ScopeScrapheap!());
    return input.strip().removechars(".,!'").toLower().split();
}
```

Fun features: Hotswapping

- 99% of the game and engine code lives in a DLL
- On successful compile, unload and reload the DLL



Hotswapping: DLL statics

- Beware DLL statics (eg. string constants)
- **Solution:** Run `dup` on all DLL statics at hotswap time
- Could be more precise, but this works fine

```
debug void RemoveRefsToStatics(T)(ref T s) if (is(T == struct))
{
    foreach (i, ref fieldExpression; s.tupleof)
    {
        enum field = __traits(identifier, s.tupleof[i]);

        static if (is(typeof(__traits(getMember, s, field))))
            RemoveRefsToStatics(__traits(getMember, s, field));
    }
}
```

```
void RemoveRefsToStatics(T)(ref T array) if (isArray!T)
{
    if (GC.addrOf(array.ptr) is null)
        array = array.dup;

    foreach (ref item; array)
        RemoveRefsToStatics(item);
}
```

Hotswapping: druntime

- DLL has a separate druntime, which isn't ideal here
- **Solution:** GC, malloc/free, threading, and other similar calls are piped back to the executable

```
struct InitGameDLLData
{
    // ...
    CAllocators cAllocators;
    FMODAllocators fmodAllocators;
    ThreadingProcs threadingProcs;
    void* physxAllocatorCallback;
    // ...
}
```

Hotswapping: vtable patching

- Be careful using classes. Vtables and classinfo might shift
- **Solution:** Limited support by manually registering classes and patching vtables on hotswap

Hotswapping: vtable patching

Startup

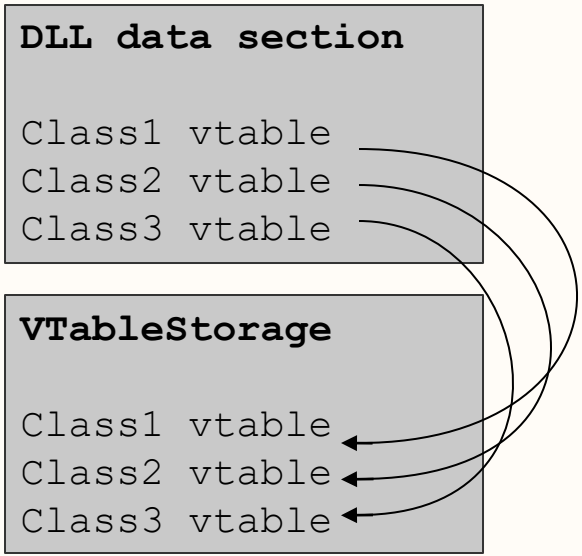
DLL data section

```
Class1 vtable
Class2 vtable
Class3 vtable
```

VTableStorage

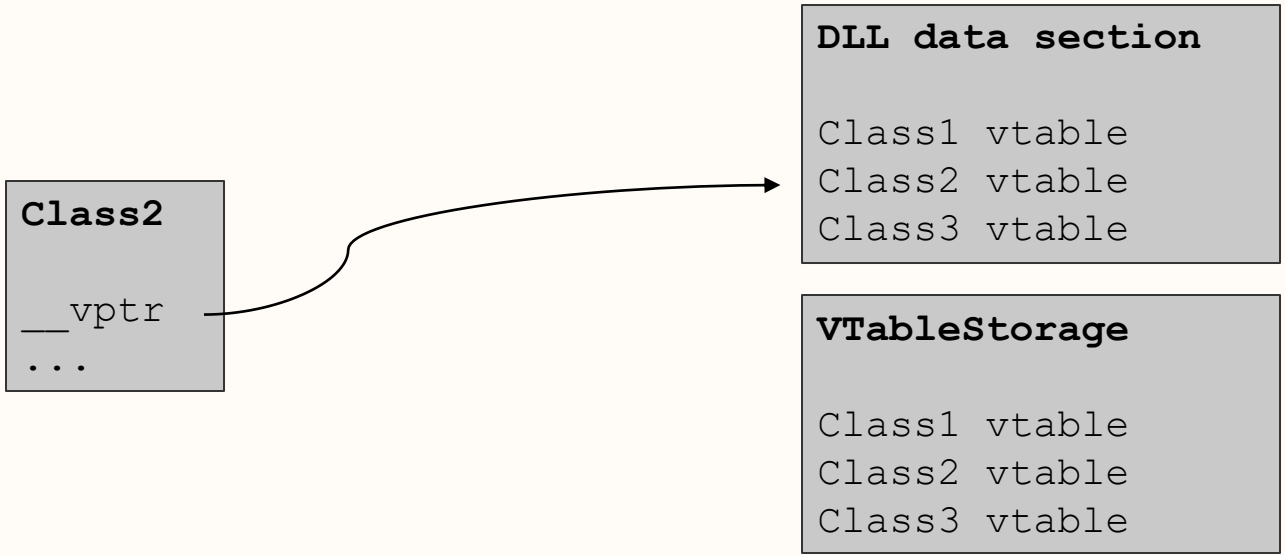
Hotswapping: vtable patching

Initialization



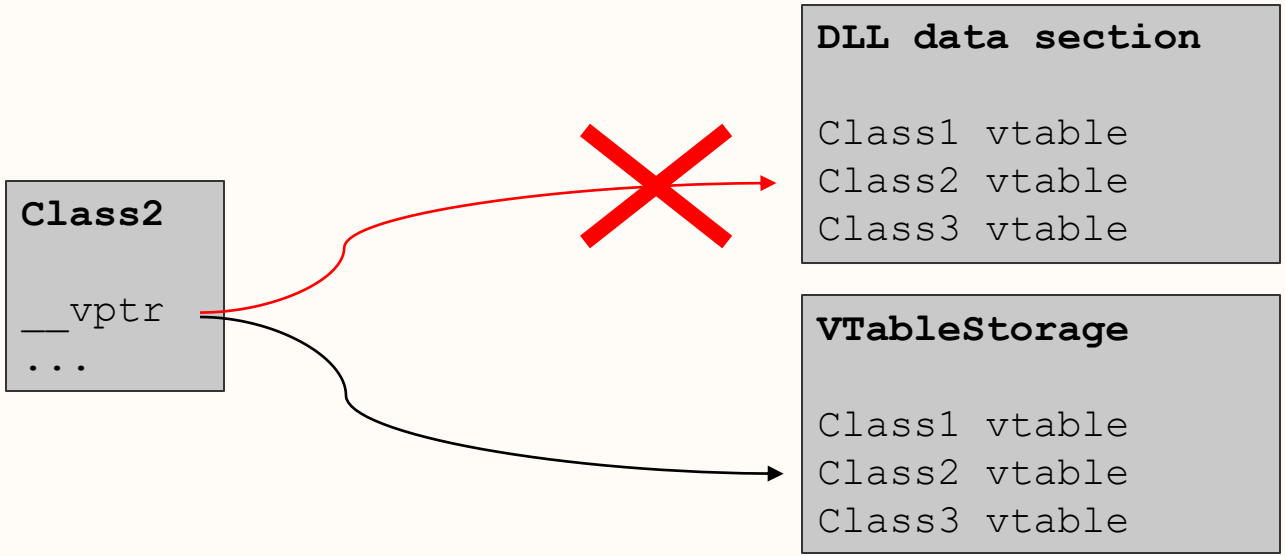
Hotswapping: vtable patching

Class2 instantiation



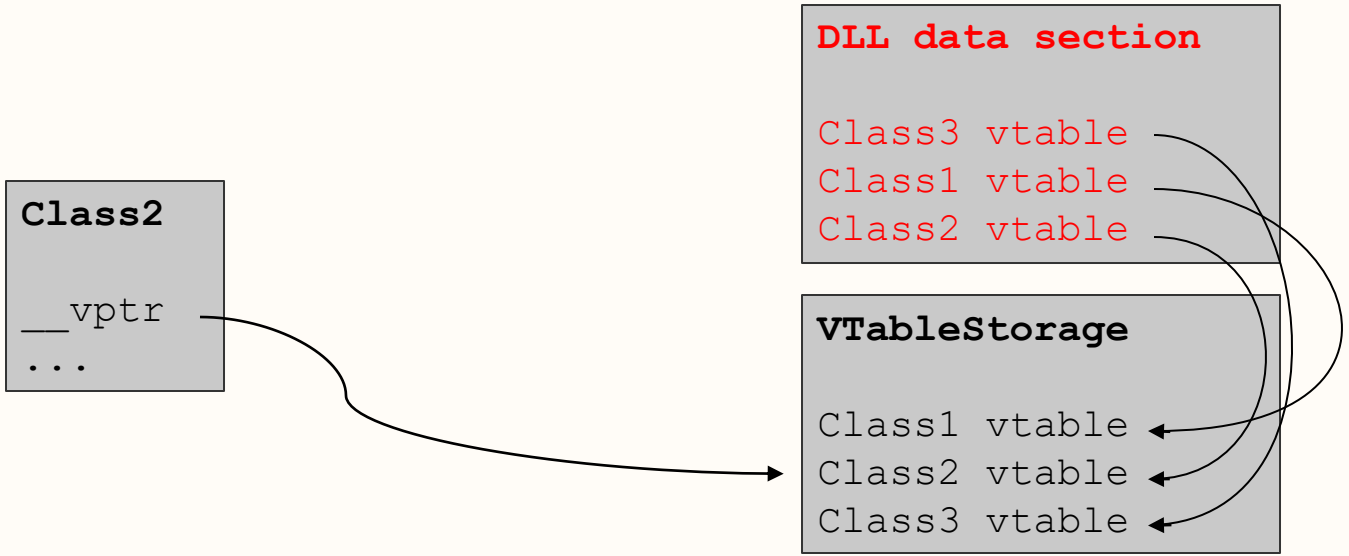
Hotswapping: vtable patching

Class2 instantiation patchup



Hotswapping: vtable patching

Hotswap



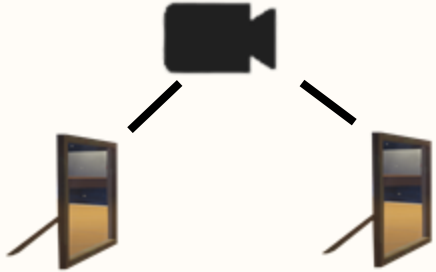
Mirror renderer: Phase 1

Pre-draw:

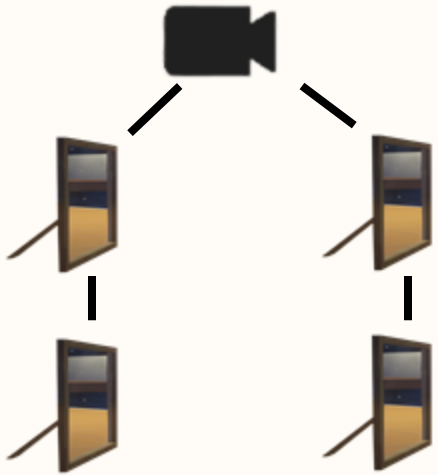
- Gather visible mirrors
- Build mirror hierarchy
- Aggressively prune the mirror hierarchy (OBB/scissor/facing checks)

Mirror tree: Depth 0

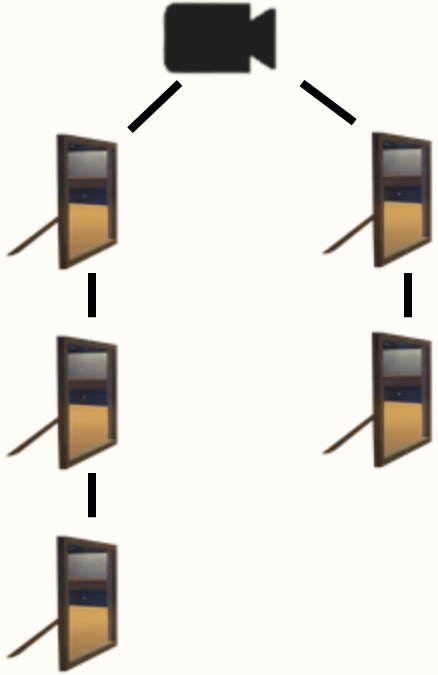
Mirror tree: Depth 1



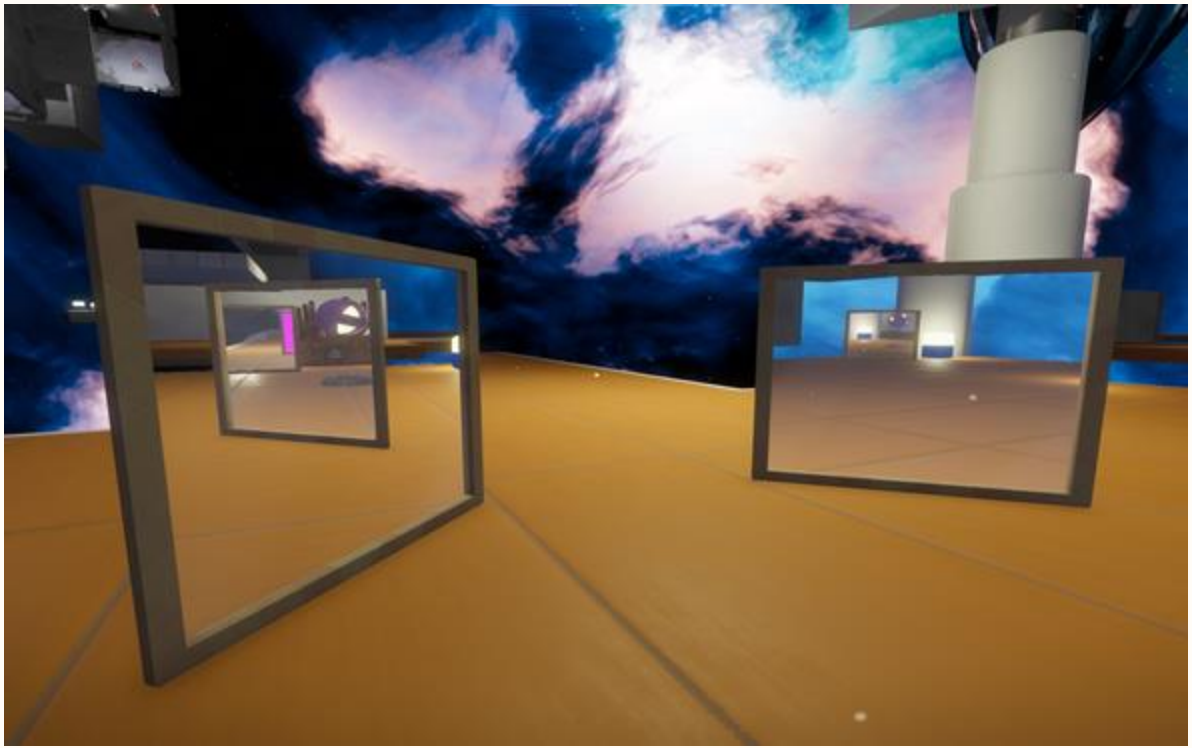
Mirror tree: Depth 2

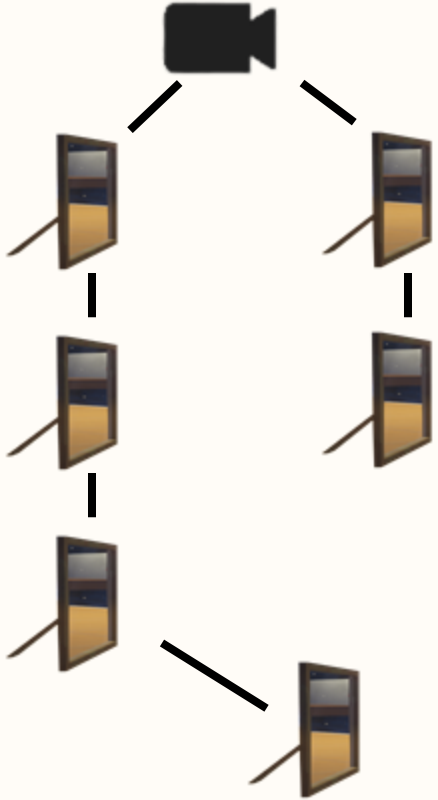


Mirror tree: Depth 3



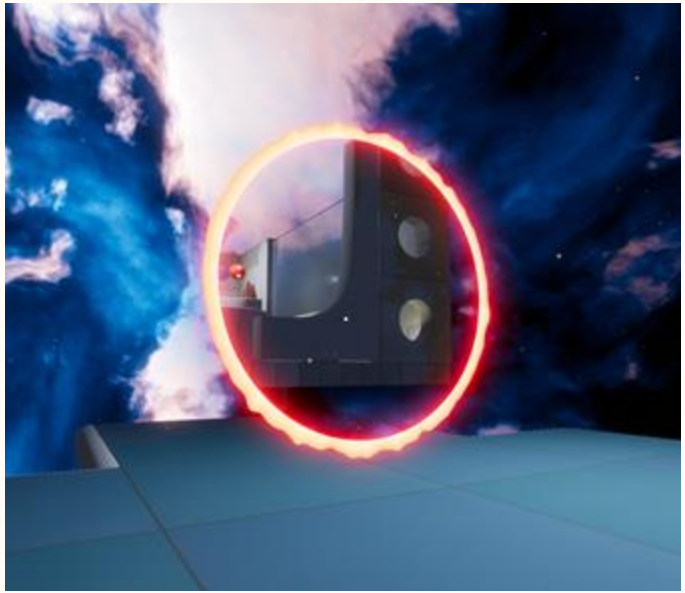
Mirror tree: Depth 4

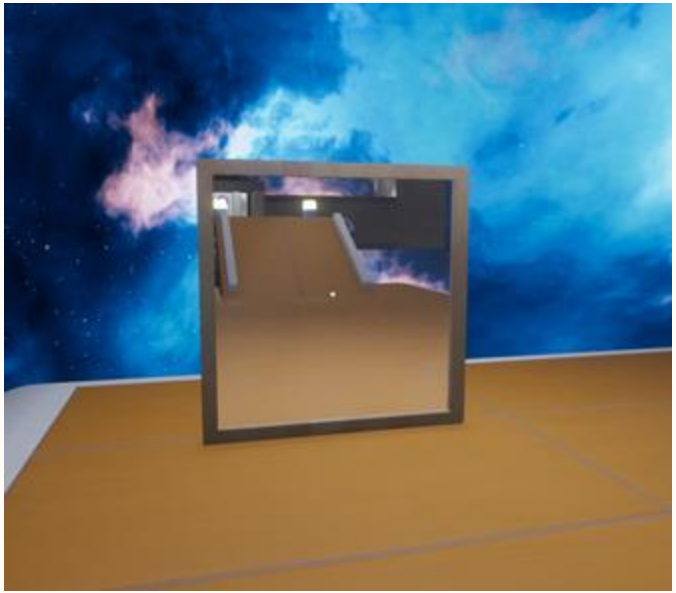




Mirror tree: Portals

Mirrors and portals are nearly identical, easy to support both

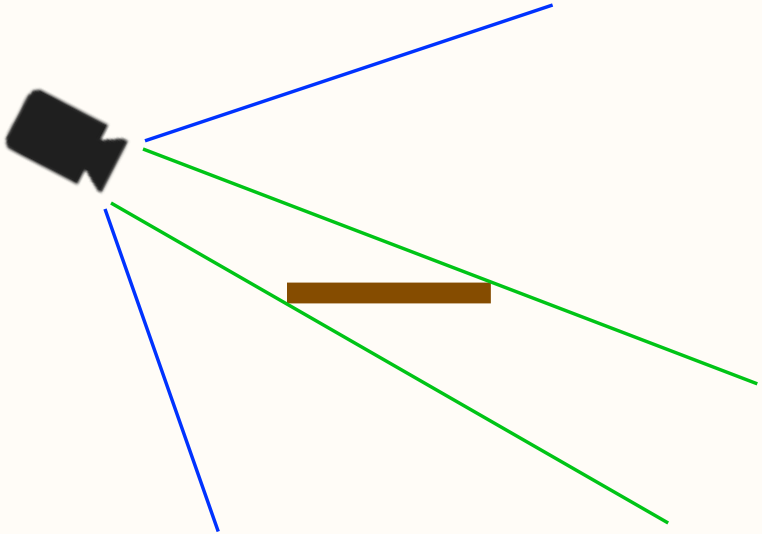




Mirror renderer: Phase 2

Parallel cull:

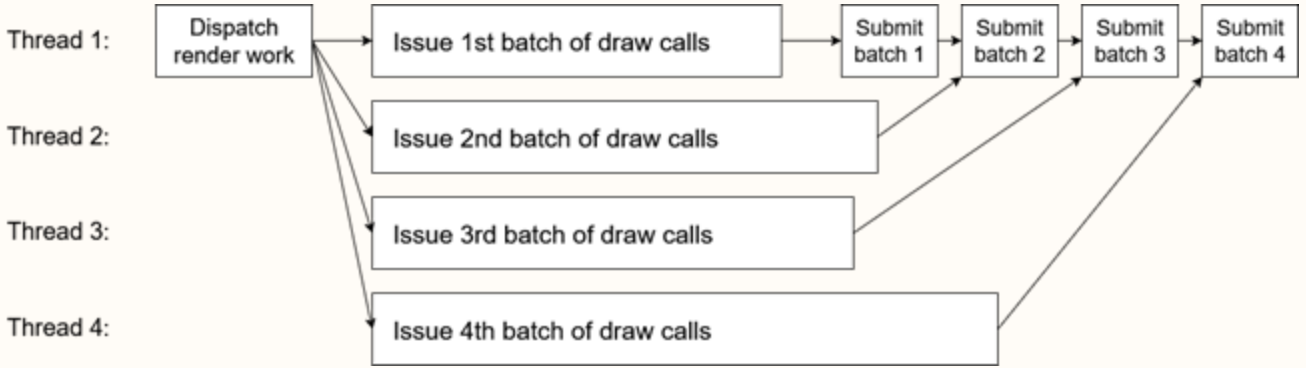
- Determine what objects each mirror can see
- Aggressively cull the visible entity list for each mirror



Mirror renderer: Phase 3

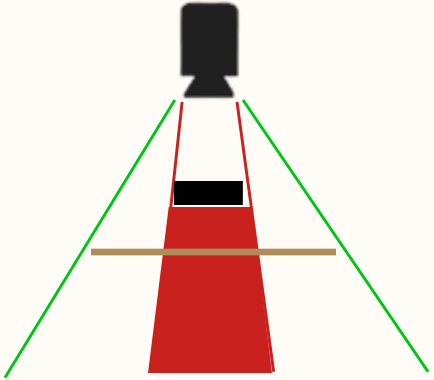
Parallel draw:

- Build command lists in parallel
- Submit sequentially on the main thread as they complete
- Coarse load balancing
- I get nearly the full 16.6ms for mirror rendering



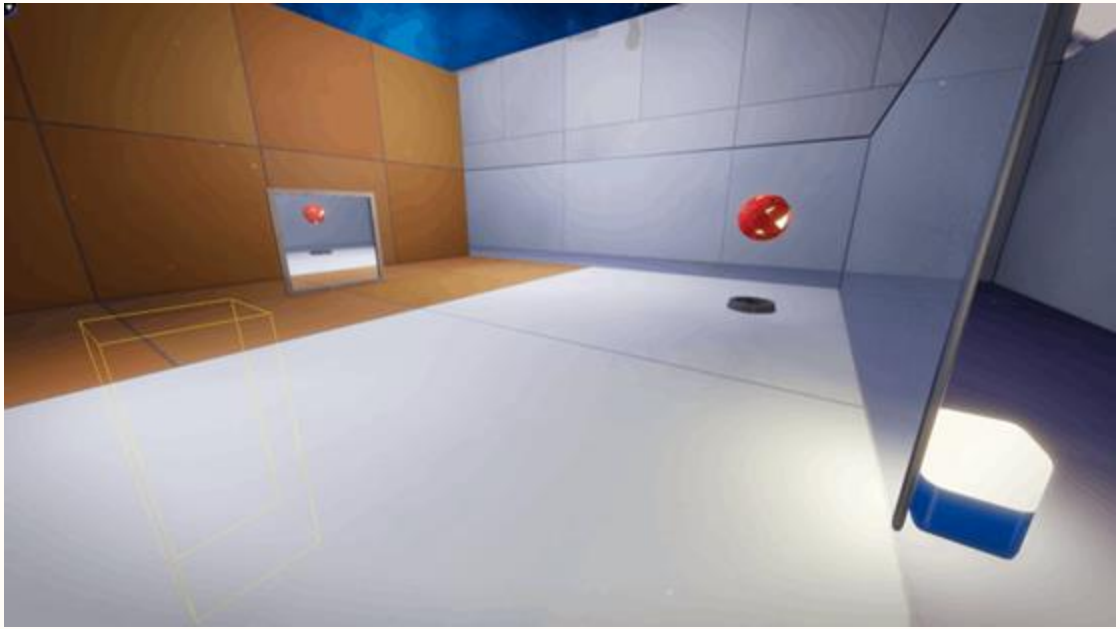
Possible performance improvements

- Occlusion culling
- Low-res CPU conservative rasterization
- Move culling and mirror pruning to the GPU
- Probably tons of room to optimize, but it's good enough today to support the game



Passing through mirrors

1. Flip screen horizontally
2. Invert controls
3. Mirror audio sources
4. Swap winding order



D libraries and bindings used

- **directx-d:** https://github.com/evilrat666/directx-d/tree/master
- **assimp:** https://code.dlang.org/packages/bindbc-assimp (no longer available?)
- **FreeType:** https://code.dlang.org/packages/bindbc-freetype
- **dear imgui:** https://code.dlang.org/packages/bindbc-imgui (plus manually added imguizmo)
- **SDL:** https://code.dlang.org/packages/bindbc-sdl
- **FMOD:** https://code.dlang.org/packages/derelict-fmod (plus my own update to FMOD 2.01)
- **gfm:** https://code.dlang.org/packages/gfm plus some of my own additions/modifications
- **msgpack:** https://code.dlang.org/packages/msgpack-d
- **fastnoise:** https://code.dlang.org/packages/fast_noise
- **PhysX:** My own hacky binding
- **SteamWorks API:** My own hacky binding
- **DirectXTex:** My own hacky binding
- **NVIDIA HBAO:** My own hacky binding

D for games: The Good

- Iteration time
- Introspection/CTFE
- Better, readable, more modifiable standard library
- Flexible and capable. Lets me do what I need to.
- So much C++ baloney fixed
- **Easy interop to existing C/C++ libraries**
- **Great Windows support**
- **Low-level access**
- Overall an obvious win over C++

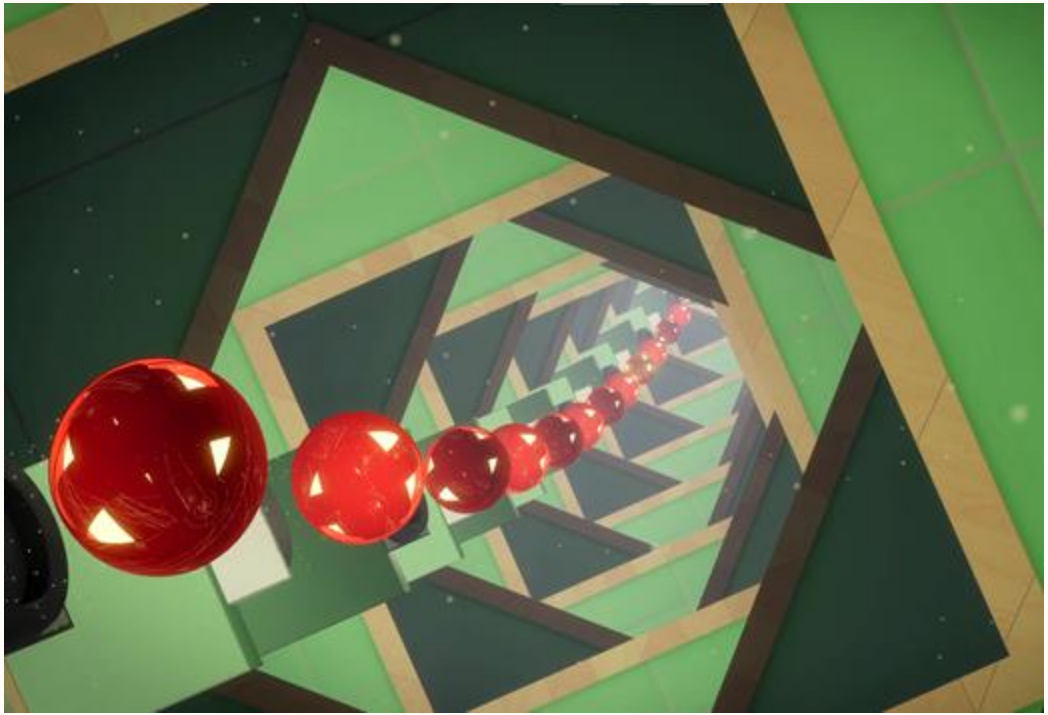
D for games: My wishlist

- Improve occasional C++ interop woes (this is hard!)
- Ecosystem size (always a tough one)
- Better support DLLs not needing their own druntime
- Occasional VS/Mago debugger weirdness

Advice for using D for games

- Do it!
- Hopefully I'm proof the language is plenty capable and mature
- You don't have to do things as weird or custom as I did

Questions?



Try the demo!

Appendix: Code hotswapping

```
LoadGameDLL(&gameState, args);

while (!shouldQuit)
{
    version (NoDLL) {}
    else
    {
        debug
        {
            SysTime throwaway;
            SysTime newDLLLastModified;
            bool gotTimes = false;
            try
            {
                getTimes("Game.dll", throwaway, newDLLLastModified);
                gotTimes = true;
            }
            catch (FileException e)
            {
                // The Game.dll file didn't exist. We might have been in the middle
                // of deleting and recreating it in our build script. Do nothing,
                // we'll probably end up reloading it next time through.
            }

                    // Reload Game.dll if necessary
            if (gotTimes && newDLLLastModified > dllLastModified)
            {
                UnloadGameDLL(gameState, false);
                LoadGameDLL(&gameState, args);
            }
        }
    }

    // Update and render the next frame
    shouldQuit = GameUpdateAndRender(gameState, timeElapsed);
}

UnloadGameDLL(gameState, true);
```

Appendix: .dup all DLL statics

```
debug void RemoveRefsToStatics(T)(ref T s)
    if (is(T == struct) && !IsManualArray!T)
{
    foreach (i, ref fieldExpression; s.tupleof)
    {
        enum field = __traits(identifier, s.tupleof[i]);
        static if (is(typeof(__traits(getMember, s, field))))
        {
            enum protection = __traits(getProtection, __traits(getMember, s, field));
            static if (protection != "private" && protection != "protected")
            {
                RemoveRefsToStatics(__traits(getMember, s, field));
            }
        }
    }
}

debug void RemoveRefsToStatics(T)(ref T array)
    if (isArray!T)
{
    static if (isDynamicArray!T)
    {
        if (array is null)
        {
            return;
        }
    }

    static if (__traits(isCopyable, ElementType!T))
    {
        if (GC.addrOf(array.ptr) is null)
        {
            array = array.dup;
        }
    }

    static if (!is(T == void[]) && !isBasicType!(ElementType!T))
    {
        foreach (ref item; array)
        {
            RemoveRefsToStatics(item);
        }
    }
}
```

Appendix: vtable patching implementation

```
// To add a class:
// 1. Add a RegisterClass mixin entry to InitGameDLL()
// 2. Add `mixin Make!MyClassName;` to the body of the class. This will disable the default constructor and create a Make()
//    function that can be called via MyClassName.Make().
// 3. If the class constructor should take args, pass those args to the Make template as well, and make your new(args)
//    implementation private.

// For things like D standard library classes where I can't easily drop in the Make() template,
// just manually call VTableManager.MakeT!T(args) instead.

template RegisterClass(T, bool isHotReload)
{
    const char[] typeName = fullyQualifiedName!T;
    const char[] strippedTypeName = fullyQualifiedName!(Unqual!T);
    const char[] vtableStr = "typeid(" ~ strippedTypeName ~ ").vtbl";
    const char[] allocationStr = isHotReload ? "" : "GS.vtableData.vtables[\"" ~ typeName ~ "\"]
        = new void*[" ~ vtableStr ~ ".length];";
    const char[] RegisterClass = allocationStr ~ "GS.vtableData.vtables[\"" ~ typeName ~ "\"][] = " ~ vtableStr ~ ";";
}

mixin template Make(T, Args...)
{
    private this() {}

    static T Make(Args...)(Args args)
    {
        import std.traits : fullyQualifiedName;
        T t = new T(args);
        debug t.__vptr = cast(immutable(void*)*) GS.vtableData.vtables[fullyQualifiedName!T].ptr;
        return t;
    }
}

T MakeT(T, Args...)(Args args)
{
    import std.traits : fullyQualifiedName;
    T t = new T(args);
    debug t.__vptr = cast(immutable(void*)*) GS.vtableData.vtables[fullyQualifiedName!T].ptr;
    return t;
}
```

Appendix: vtable patching usage

```
void InitGameDLL(bool isHotReload)
{
    // ...
    mixin(VTableManager.RegisterClass!(shared Mutex, isHotReload));
    mixin(VTableManager.RegisterClass!(PhysicsSystem.ErrorCallback, isHotReload));
    mixin(VTableManager.RegisterClass!(EditorCommands.EditorCommand, isHotReload));
    // ...
}
```

Appendix: GameState Inspector

```
void igEditStructHelper(T)(ref T var, string name, int depth, ref uint curFieldID, bool showAllMembers, bool doIndent = true)
{
    static if (is(T == struct))
    {
        if (depth == 0 || igTreeNodeStr(name.toStringz()))
        {
            GroupPrologue(depth);

            foreach (i, ref fieldExpression; var.tupleof)
            {
                enum field = __traits(identifier, var.tupleof[i]);
                static if (is(typeof(__traits(getMember, var, field))))
                {
                    enum protection = __traits(getProtection, __traits(getMember, var, field));
                    static if (protection != "private" && protection != "protected")
                    {
                        igEditStructHelper(__traits(getMember, var, field), field, depth + 1, curFieldID, showAllMembers);
                    }
                }
            }

            GroupEpilogue(depth);
        }
        else
        {
            igSpacing();
        }
    }
    else static if (isSomeString!T || isConvertibleToString!T)
    {
        if (doIndent) igIndent();
        igPushItemWidth(300);

        const char* cstring = var.toStringz();
        igInputText(EditStructLabel(name, curFieldID), cast(char*) cstring, var.length + 1, ImGuiInputTextFlags_ReadOnly);

        igPopItemWidth();
        if (doIndent) igUnindent();
    }
    // …
}
```

Appendix: StringHash.exe

```
__gshared auto SID_REGEX = ctRegex!(`\bSID\s*\(\s*"(.*?)(?<!\\)"\s*?(,\s*0x[a-f0-9]{1,8})?\s*\)`);

foreach (DirEntry entry; dirEntries("src", SpanMode.breadth))
{
    if (!entry.isFile) continue;

    bool foundSID = false;
    string[] lines = null;
    File file = File(entry.name, "r");
    string curLine = file.readln();
    int lineNumber = 1;
    while (curLine !is null)
    {
        string modifiedLine = curLine;
        int insertOffset = 0;
        foreach (match; curLine.matchAll(SID_REGEX))
        {
            uint hash = cast(uint) match[1].hashOf();
            string stringToInsert = format("SID(\"%s\", 0x%x)", match[1], hash);
            int replacePoint = (match[0].ptr - curLine.ptr) + insertOffset;
            int replaceEnd = replacePoint + match[0].length;
            modifiedLine = modifiedLine[0..replacePoint] ~ stringToInsert ~ modifiedLine[replaceEnd..$];
            insertOffset += stringToInsert.length - match[0].length;
            foundSID = true;
        }

        lines ~= modifiedLine;
        curLine = file.readln();
        lineNumber++;
    }

    if (foundSID)
    {
        file = File(entry.name, "w");
        foreach (line; lines)
        {
            file.write(line);
        }
    }
}
```

```
if (myID == SID("SomeStringID"))
```

```
if (myID == SID("SomeStringID", 0x1de87b34))
```