Frictionless Debug UI in C++

What I Want to Talk About

- Introduce Dear IMGui

- Show some tricks I’m using in some tools I’m experimenting with

Friction?

The amount of busy work needed to get something done, that isn't directly related to what you want to do.

Example:

You want add a tweakable parameter to an algorithm you’re working on.

But doing this may also require:

- Updating the header
- Updating comments / documentation
- Updating UI
- Updating serialization
- Waiting for things to compile and load
- etc.

Why Care About Friction?

- Takes time
- Reduces momention and train of thought
- Makes people more likely to make “lazy” choices
- Takes the joy out of development

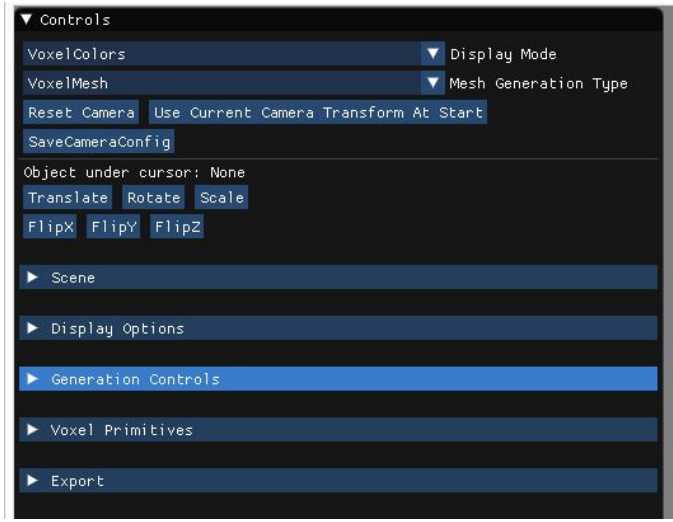
How to Reduce Friction

- Evaluate your processes and coding conventions
- Defer the work until you know it’s needed
- Automation
- Try to keep compile times and loading times low
- Make it so you can tweak things without re-compiling and loading
- Keep duplication to a minimum
- etc.

Debug and Tool UI

Tools and debug UI can make parts of the development process way easier

Unfortunately, it also increases the amount of stuff you need to update and maintain



Usually, we don’t care as much about how it looks as long as it’s functional.

What is an Immediate Mode UI

- Similar to OpenGL immediate mode
  - (Note: Dear ImGui doesn't use immediate mode rendering)
- Basically with IM GUIs the controlling code also creates the UI.
- There is very little state to maintain with and IM Gui.
- Unity has their own version of an Immediate Mode Gui.

Retained vs Immediate Mode UI

Retained

- UI layout is separate from the logic controlling it.
- Tool assisted layouts are easier and more common.
- Easier to do advanced visuals like animations.
- Adding to the UI often involves creating separate UI elements and extra “glue code”
- More likely for UI and logic to get out of sync.

Immediate Mode

- UI layout is generated from the controlling logic.
- Layouts are more automatic, but harder to customize.
- Advanced visuals are more difficult.
- Less extra code and data is needed to make the UI work.
- Less likely for the UI to get out of sync.

Dear IMGui

An easy to use immediate mode GUI library for C++.

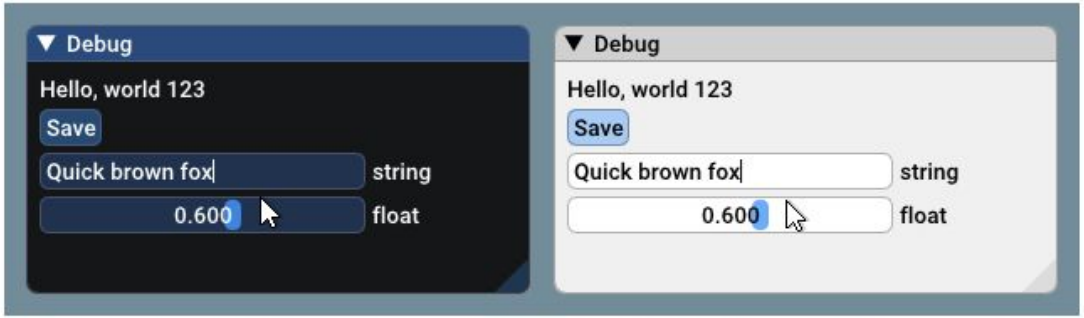
- Minimal dependencies
- Can be integrated into existing engines without too much trouble
- Should work with any graphics library.

Example Usage

Code:

```
ImGui::Text("Hello, world %d", 123);
if (ImGui::Button("Save"))
{
    // do stuff
}
ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));
ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```

Result:



(settings: Dark style (left), Light style (right) / Font: Roboto-Medium, 16px / Rounding: 5)

Getting Set Up with ImGUI

To make Dear IMGui work you need to:

- Pass input and settings to Dear ImGUI

- Setup a render callback to render the draw data

Getting Set Up with ImGUI

I’m using SFML (Simple and Fast Multimedia Library) for my tools.

I referred to this link to help get set up:

https://eliasdaler.github.io/using-imgui-with-sfml-pt1/

This link provides some binding code to help get set up:

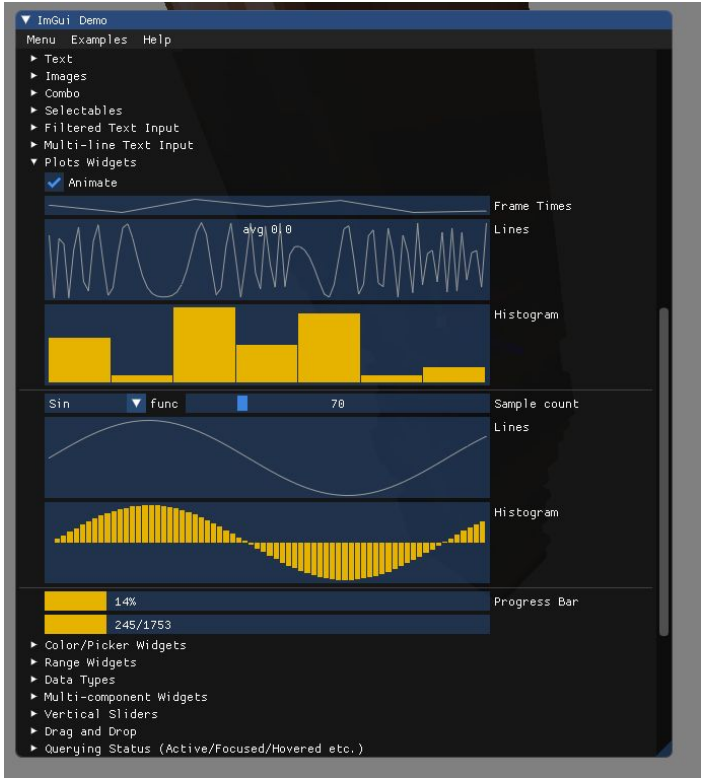
https://github.com/eliasdaler/imgui-sfml

Tricks for Learning Dear ImGUI features

There’s a handy function called:

```
ImGui::ShowTestWindow();
```

Browsing through this window and code is usually enough to figure out new features, and is often faster than Google

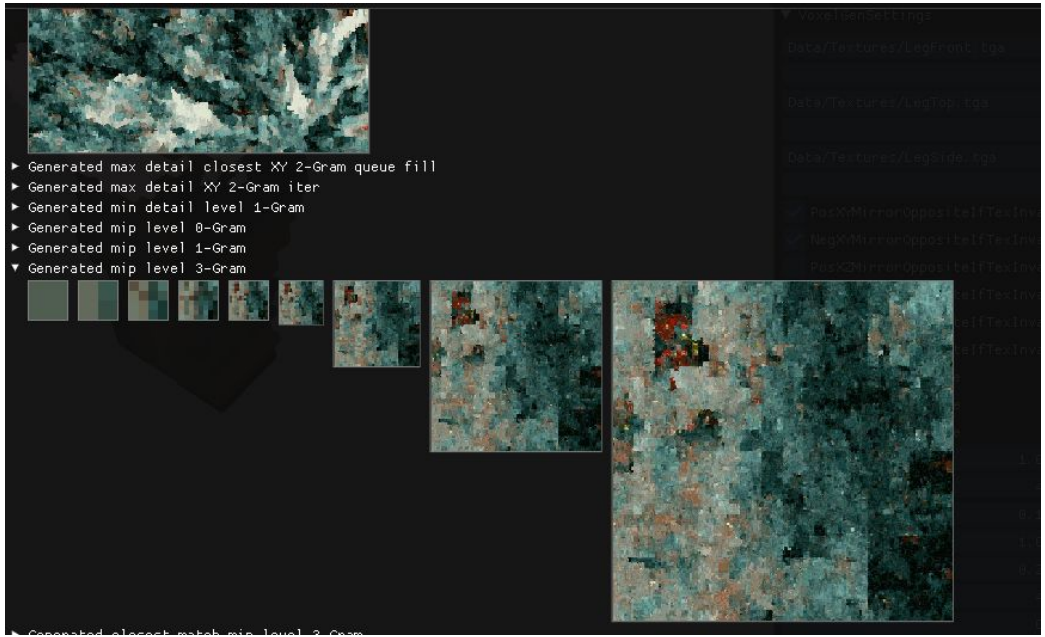


Examples from My Tools

Drag and drop scene hierarchy:

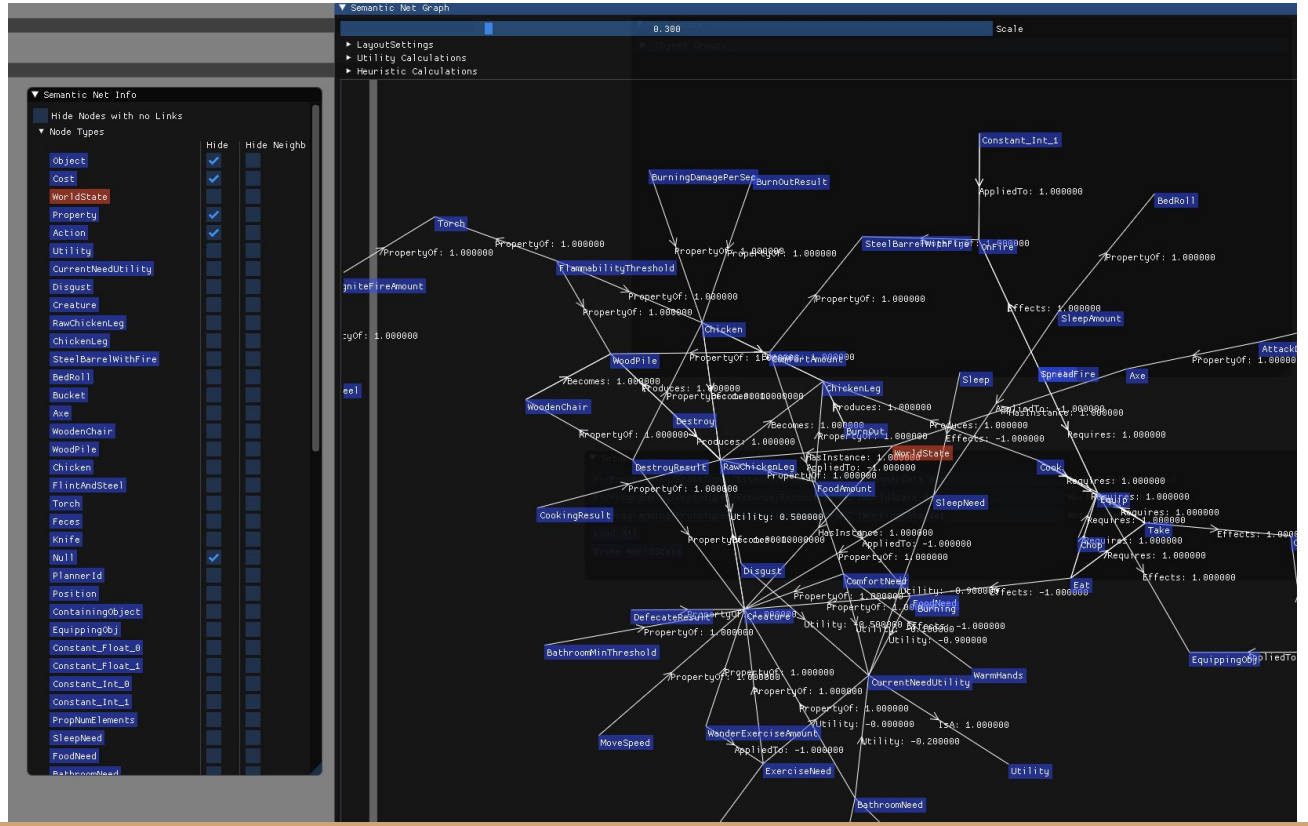
Examples from My Tools

Displaying Images:



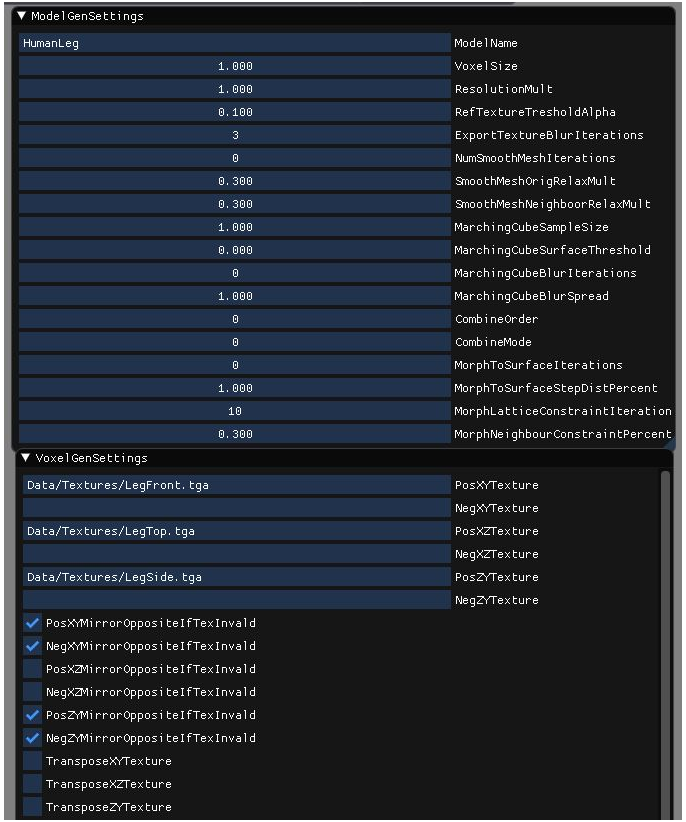
Examples from My Tools

Displaying Graphs:



Examples from My Tools

Macro generated UI for tweaking properties:



Code Generation using Macros

- Even with IMGui there is still extra code to maintain when adding new properties.

- Ideally I’d like the UI to update automatically when I change properties.

- I’m using a macro trick to generate the UI automatically.

Cool Macro Features I Didn't Learn in School

Turning macro parameters into strings using the stringizing operator (#)

```
// stringizer.cpp
#include <stdio.h>
#define stringer( x ) printf_s( #x "\n" )
int main() {
   stringer( In quotes in the printf function call );
   stringer( "In quotes when printed to the screen" );
   stringer( "This: \"  prints an escaped double quote" );
}
```

```
In quotes in the printf function call
"In quotes when printed to the screen"
"This: \"  prints an escaped double quote"
```

Cool Macro Features I Didn't Learn in School

Concatenating / merging arguments using the token-pasting operator (##)

```
// preprocessor_token_pasting.cpp
#include <stdio.h>
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;

int main()
{
   paster(9);
}
```

```
token9 = 9
```

Cool Macro Features I Didn't Learn in School

You can undefine / redefine macros:

```
#ifdef E_PROP

#error E_PROP is already defined

#endif

#define E_PROP(name, type, luaType, defaultValue, minValue, maxValue, updateFlags) type name
= defaultValue;

//Do stuff

#undef E_PROP
```

Generating Code Based on Lists of Stuff

Using a combination of inline files and macro definitions you can generate code based on lists of macro calls.

This can be used to:

- Define enums and string conversion functions
- Generate UI to display and tweak properties
- Generate serialization code for properties
- Generate script binding code
- etc.

Generating Code Based on Lists of Stuff

Define an inline file with macro calls (This example defines an enum)

```
E_ENUM(Set)             //Dest = Source
E_ENUM(Add)             //Dest += Source
E_ENUM(Subtract)        //Dest -= Source
E_ENUM(Create)          //Create Object based on Source prop
E_ENUM(Destroy)         //Destroy Dest Object
E_ENUM(ApproachTarget)  //
```

Generating Code Based on Lists of Stuff

Define what the macro should do then include the inline file:

```
E_ENUM(Set)             //Dest = Source
E_ENUM(Add)             //Dest += Source
E_ENUM(Subtract)        //Dest -= Source
E_ENUM(Create)          //Create Object based on Source prop
E_ENUM(Destroy)         //Destroy Dest Object
E_ENUM(ApproachTarget)  //
```

```
enum MathFunctionType
{
    #ifdef E_ENUM
    #error E_ENUM is already defined
    #endif

    #define E_ENUM(mode) mode,
    #include "MathFunctionTypes.inl"
    #undef E_ENUM
};
```

Generating Code Based on Lists of Stuff

Define what the macro should do then include the inline file:

```
E_ENUM(Set)             //Dest = Source
E_ENUM(Add)             //Dest += Source
E_ENUM(Subtract)        //Dest -= Source
E_ENUM(Create)          //Create Object based on Source prop
E_ENUM(Destroy)         //Destroy Dest Object
E_ENUM(ApproachTarget)  //
```

```
const char *MathFunctionTypes::GetString(int mode)
{
    switch (mode)
    {
#ifdef E_ENUM
#error E_ENUM is already defined
#endif

#define E_ENUM(mode) case mode: return #mode; \
        break;

#include "MathFunctionTypes.inl"
#undef E_ENUM
    }

    E_DEBUG_ERROR(L"Invalid mode: %d", mode);
    return "<INVALID FUNCTION>";
}
```

Generating Code Based on Lists of Stuff

Define what the macro should do then include the inline file:

```
MathFunctionTypes::MathFunctionType MathFunctionTypes::GetType(const char* typeStr)
{
    #ifdef E_ENUM
    #error E_ENUM is already defined
    #endif

    #define E_ENUM(type) if (strcmp(typeStr, #type) == 0) \
    {\
        return type;\
    }
    #include "MathFunctionTypes.inl"
    #undef E_ENUM

    E_DEBUG_ERROR(L"Type not found for: %S", typeStr);

    return AlwaysZero;
}
```

Macro Based UI and Serialization Generation

(let's look at the code for this)

Links

Dear IMGui Github page:

https://github.com/ocornut/imgui

Using Dear ImGui with SFML

https://eliasdaler.github.io/using-imgui-with-sfml-pt1/