Procedural Texture Generation Tool in C++

Review for BcnCppProgrammers Meetup

Image from *Where the dead things dwell* 8k Intro by Architect

Who I am

David Gallardo Moreno

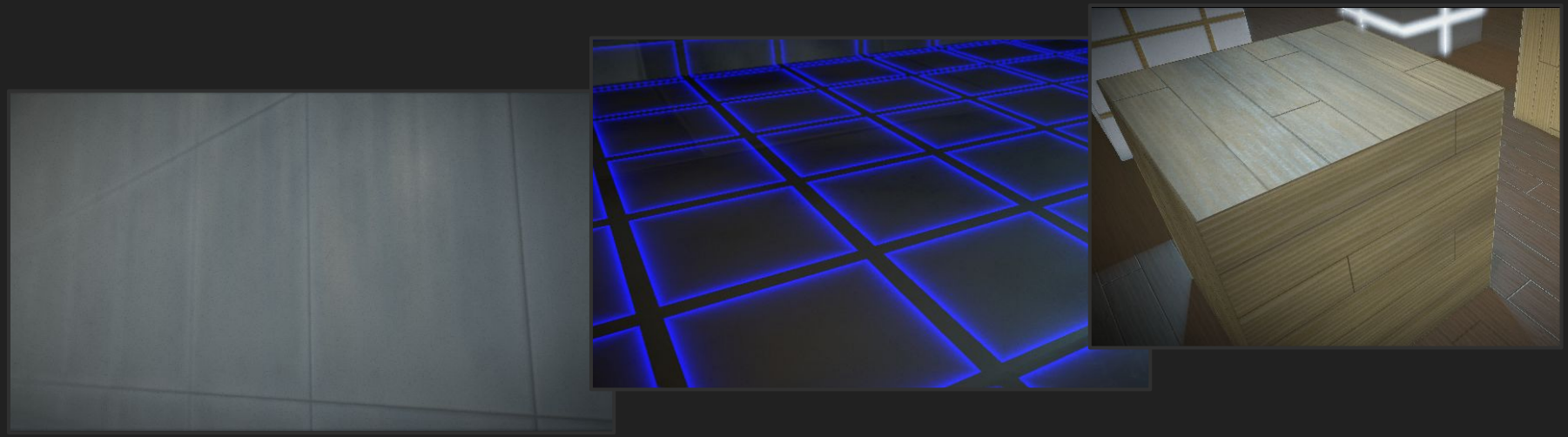
Lead Programmer (Afterpulse) at Digital Legends Entertainment (Mobile)
Past: Gameplay Programmer at Virtual Toys (PS4) and Bee Square (Mobile).

Twitter: @galloscript

I do things with C++ and Graphics in my free time.

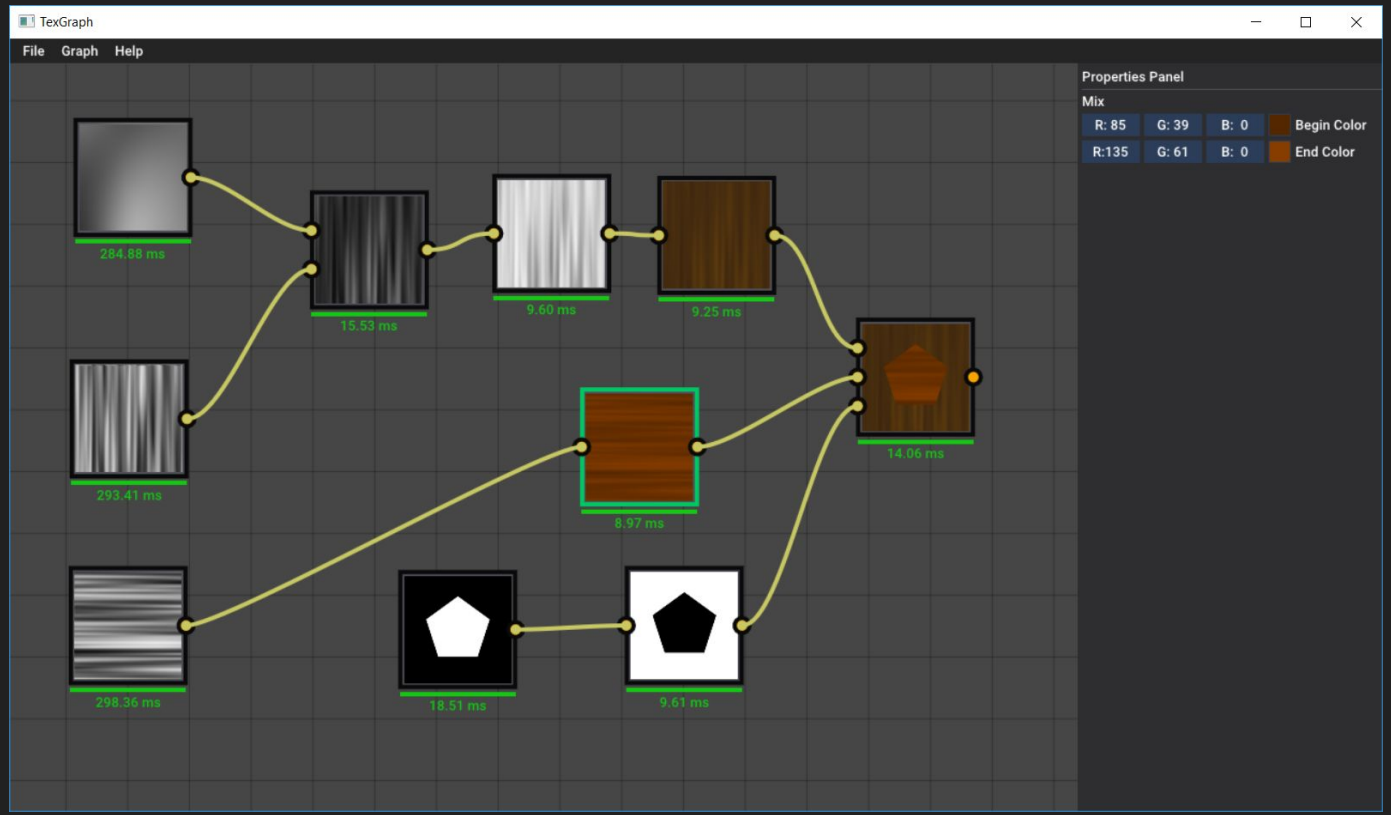
Why this tool?

- Have something similar to industry standard tools to generate textures for a 8k intro (demoscene).
- Editing code and running on GPU to see the results works, but it’s a tedious work and a waste of time in trial and error.



Procedural Materials from *Where the dead things dwell* 8k Intro by Architect

Let’s see the tool (Demo)



Designing Node Graph with Inheritance (FAIL)

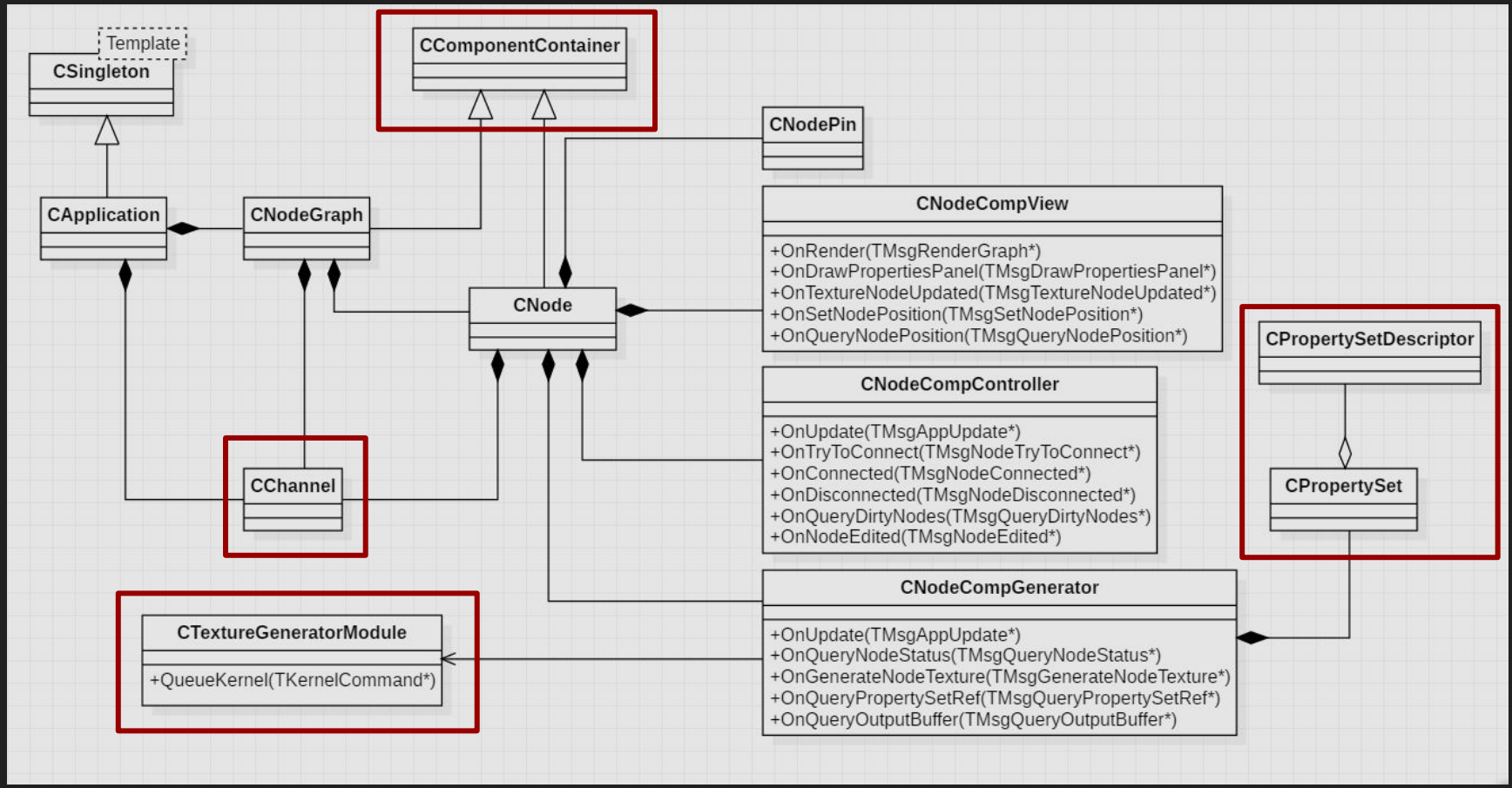
First approach: Base Node class with virtual functions Update(), Draw(), Generate(), Serialize(), etc.

- Every new type of Node requires a new class.
- Nodes can not be defined with data.
- Adding new functionality may require to modify all the children classes.
- Highly coupled implementation.
- Hard to maintain and to add new Nodes.

Decoupled data driven Nodes

- Use **components** over inheritance.
- Use **observer pattern** to communicate objects whenever it makes sense.
- Use **dynamic data structures** to store specific Node properties
- Implement a **standalone texture generation** module.
- Implement a **factory** that knows the **components** and **properties** of every Node to create it with data.

Node Graph Class Diagram



Components

In this implementation ...

- A **component is an object** that adds functionality to a **component container.**
- All Nodes have the same 3 components: View, Controller, and Generator.
- Doesn’t inherit any specific class.
- Are deleted when it’s container is deleted.
- Know about it’s container interface but not about other components.
- There can be only **one instance for each type** of component in a container.

Component Container

```cpp
namespace __component_deleter
{
    template <typename T> void ComponentDeleter(void* aObject)
    {
        SBX_DELETE(reinterpret_cast< T* >(aObject));
    }
}

class CComponentContainer
{
private:
    struct TStoredComponent
    {
        typedef void(*TComponentDeleterFnc)(void*);

        void* mObject;
        TComponentDeleterFnc mDeleter;

        TStoredComponent(void* aObject, TComponentDeleterFnc const & aDeleter)
            : mObject(aObject)
            , mDeleter(aDeleter)
        {
        }
    };
public:
                                CComponentContainer ();
    virtual                    ~CComponentContainer ();
    template <typename T> T*    AddComponent        (T* aComponent);
    template <typename T> bool  RemoveComponent     ();
    template <typename T> T*    GetComponent        ();
private:
    std::map<uint32_t, TStoredComponent> mComponents;
};
```

Knows how to delete the component

```cpp
template <typename T> T* CComponentContainer::AddComponent(T* aComponent)
{
    const uint32_t lTypeId = GetTypeId<T>();
    auto lIterator = mComponents.find(lTypeId);
    if(lIterator == mComponents.end())
    {
        mComponents.insert(lIterator,
                           std::make_pair
                           (
                               lTypeId,
                               TStoredComponent
                               (
                                   aComponent,
                                   &__component_deleter::ComponentDeleter<T>
                               )
                           ));
        return aComponent;
    }

    SBX_ERROR("Can't add the same component type twice.");
    return nullptr;
}
```

Check if the component exist and if not, store the instance with it's *deleter*

```cpp
CNodeGraph* lGraph = new CNodeGraph();
CNode* lNode = new CNode(lGraph);
lNode->AddComponent(new CNodeCompView(lNode));
lNode->AddComponent(new CNodeCompController(lNode));
lNode->AddComponent(new CNodeCompGenerator(lNode));

CNodeCompView* lView = lNode->GetComponent< CNodeCompView >();

lNode->RemoveComponent< CNodeCompView >();
```

Usage

Messaging System with Channels

- A **Channel** is an object that maps callbacks (observers) with a **message** type.
- Every type of **message** is a struct.
- Channels calls every registered observer with a pointer to the message instance we are sending.
- In this implementation, there is a channel for:
    - Every node instance
    - Graph
    - Gui System
    - Application

Channel Interface

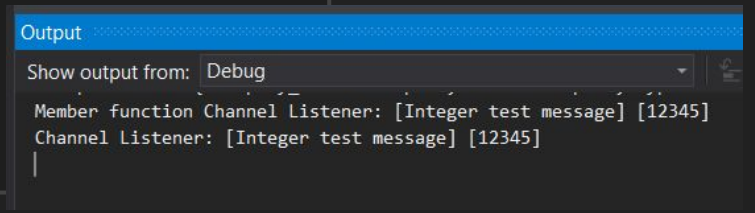
```
class CChannel
{
public:
                                        CChannel                ();
                                       ~CChannel                ();
    void                                UnregisterAllCallbacks  ();

    template <typename T> void              RegisterCallback    ( void(*aCallback)(T const *) );
    template <typename T, typename U> void  RegisterCallback    (U* aObject, void(U::*aCallback)(T const *) );

    template <typename T> void              UnregisterCallback  ( void(*aCallback)(T const *) );
    template <typename T, typename U> void  UnregisterCallback  (U* aObject, void(U::*aCallback)(T const *) );

    template <typename T> void              BroadcastMessage    (T const & aMsg) const;

private:
    typedef std::map< uint32_t, _impl::TStoredBroadcaster >::iterator TBroadcasterIt;
    template <typename T> TBroadcasterIt    GetOrCreateBroadcaster  ();

private:
    std::map< uint32_t, _impl::TStoredBroadcaster > mBroadcasters;
};
```

The map contains a unique message type (T) identifier as key, and a vector of callback bindings as value

Channel Usage Example

```
sbx::CChannel lChannel;

struct TestListener
{
    void MyListenerFnc(const TMsgTest<int>* aMsg)
    {
        WriteLog("Member function Channel Listener: [%s] [%d]", aMsg->mText.c_str(), aMsg->mValue);
    }
};

//register memeber function callback
TestListener* lTestListener = new TestListener;
lChannel.RegisterCallback( lTestListener, &TestListener::MyListenerFnc );

auto lFnc = [](const TMsgTest<int>* aMsg)
{
    WriteLog("Channel Listener: [%s] [%d]", aMsg->mText.c_str(), aMsg->mValue);
};

//register lambda callback
lChannel.RegisterCallback< TMsgTest<int> >(lFnc);

//broacast message
lChannel.BroadcastMessage( TMsgTest<int>("Integer test message", 12345) );
lChannel.UnregisterCallback( lTestListener, &TestListener::MyListenerFnc );
lChannel.UnregisterCallback< TMsgTest<int> >(lFnc);
```

```
template <typename T> struct TMsgTest
{
    std::string mText;
    T mValue;

    TMsgTest(const char* aText, T aValue = T())
        : mText(aText)
        , mValue(aValue)
    {;}
};
```



Property Set: Dynamic Data Structures

- A Property set is a buffer + a pointer to a descriptor.
- A Descriptor contains the name, offset and size of all the set properties.
- It’s purpose is to define and hold datagrams with data, not code.
- Each Node have a property set in it’s Generator component.
- The property set is shared with other components and modules.

Property Set Descriptor Interface

Stores all the required information to iterate the properties.

```
struct TPropertyInfo
{
    TUniqueIdCS     mName;
    TUniqueIdCS     mDisplayName;
    EPropertyType   mType;
    uint32_t        mDataOffset;
    uint32_t        mSize;
    any             mDefaultValue;
    any             mMinValue;
    any             mMaxValue;
};
```

```
class CPropertySetDescriptor
{
public:
                            CPropertySetDescriptor();
                           ~CPropertySetDescriptor();
    bool                    Init                    (TPropertyInfo* aPropertiesInfo, uint32_t aPropertiesCount);
    uint32_t                GetPropertyInfoCount    () const;
    const TPropertyInfo*    GetPropertyInfoAt       (uint32_t aIndex) const;
    uint32_t                GetPropertyIndex        (TUniqueIdCS const & aUniqueId) const;

private:
    std::vector< TPropertyInfo* >   mPropertiesInfoList;
};
```

Property Set Interface

Contains the buffer with all properties values and the interface to iterate and read / write them.

```
class CPropertySet
{
    SBX_DISALLOW_COPY(CPropertySet)
public:
                                CPropertySet            ();
                               ~CPropertySet            ();
    bool                        Init                    (CPropertySetDescriptor const *aDescriptor);
    uint32_t                    GetPropertyInfoCount    () const;
    const TPropertyInfo*        GetPropertyInfoAt       (uint32_t aIndex) const;
    uint8_t*                    GetPropertyRawDataPtrAt (uint32_t aIndex);
    const uint8_t*              GetPropertyRawDataPtrAt (uint32_t aIndex) const;
    uint32_t                    GetPropertyIndex        (TUniqueIdCS const & aUniqueId) const;

    template <typename T> T*    GetPropertyDataPtrAt    (uint32_t aIndex);
    template <typename T> T     GetPropertyValue        (TUniqueIdCS const & aUniqueId) const;
    template <typename T> T     GetPropertyValue        (uint32_t aIndex) const;
    template <typename T> void  SetPropertyValue        (TUniqueIdCS const & aUniqueId, T aValue);
    template <typename T> void  SetPropertyValue        (uint32_t aIndex, T aValue);
private:
    const CPropertySetDescriptor*   mDescriptor;
    uint8_t*                        mPropertiesData;
};
```

Property Set usage: Mix node

```
{
    "name": "Mix",
    "display_name": "Mix",
    "kernel": "mix",
    "components": [ "NodeCompView", "NodeCompController", "NodeCompGenerator" ],
    "color": [ 100, 100, 0 ],
    "input_pins": [
        {
            "name": "input",
            "display_name": "Input",
            "type": "R"
        }
    ],
    "output_pins": [
        {
            "name": "output",
            "display_name": "Output",
            "type": "RGB"
        }
    ],
    "properties": [
        {
            "name": "begin",
            "display_name": "Begin Color",
            "type": "Color",
            "default_value": [ 0, 150, 0 ]
        },
        {
            "name": "end",
            "display_name": "End Color",
            "type": "Color",
            "default_value": [ 0, 0, 200 ]
        }
    ]
},
```

Definition of a property set descriptor with data, later used to create the buffer and initialize it.

```
bool CPropertySet::Init (CPropertySetDescriptor const *aDescriptor)
{
    mDescriptor = aDescriptor;

    if(mPropertiesData)
    {
        SBX_FREE(mPropertiesData);
        mPropertiesData = 0;
    }

    uint32_t lTotalSize = 0;
    for(uint32_t i = 0; i < GetPropertyInfoCount(); ++i)
    {
        const TPropertyInfo* lProperty = GetPropertyInfoAt(i);
        lTotalSize += lProperty->mSize;
    }

    mPropertiesData = (uint8_t*)SBX_MALLOC(lTotalSize);
    ::memset(mPropertiesData, 0, lTotalSize);

    for(uint32_t i = 0; i < GetPropertyInfoCount(); ++i)
    {
        const TPropertyInfo* lProperty = GetPropertyInfoAt(i);
        ::memcpy( &mPropertiesData[lProperty->mDataOffset],
                  &lProperty->mDefaultValue,
                  lProperty->mSize);
    }
    return true;
}
```

Property Set usage: Mix Kernel

Retrieve values from the property set and store for later use in the execution

```
struct TKernelMix : public TKernelCommand
{
    glm::vec3 mBeginColor;
    glm::vec3 mEndColor;

    TKernelMix(CPropertySet const * aPropertySet)
    {
        const TColor lBeginColor = aPropertySet->GetPropertyValue< TColor >(uidcs(begin));
        const TColor lEndColor   = aPropertySet->GetPropertyValue< TColor >(uidcs(end));
        mBeginColor = glm::vec3(lBeginColor.r, lBeginColor.g, lBeginColor.b);
        mEndColor   = glm::vec3(lEndColor.r, lEndColor.g, lEndColor.b);
    }

    void Execute(int32_t x, int32_t y) override
    {
        const int32_t lInputCoordX = int32_t((x / float(mOutputBuffers[0].GetWidth())) * mInputBuffers[0].GetWidth());
        const int32_t lInputCoordY = int32_t((y / float(mOutputBuffers[0].GetHeight())) * mInputBuffers[0].GetHeight());
        const TColor    InputColor = mInputBuffers[0].GetColor(lInputCoordX, lInputCoordY);
        const glm::vec3 lVec3Color = glm::mix(mBeginColor, mEndColor, glm::vec3(InputColor.r, InputColor.g, InputColor.b));
        mOutputBuffers[0].SetColor(x, y, TColor(lVec3Color.x, lVec3Color.y, lVec3Color.z));
    }
};
```

Full Example: Properties panel (1 / 2)

```
void CNodeCompView::OnDrawPropertiesPanel(const TMsgDrawPropertiesPanel* aMsg)
{
    ImGui::Text("%s", mData->mNode->GetName().GetStr());

    TMsgQueryPropertySetRef lQueryPropertySetRef;
    mData->mNode->GetChannel()->BroadcastMessage(lQueryPropertySetRef);
    if(lQueryPropertySetRef.mPropertySet)
    {
        bool lEdited = false;
        CPropertySet* lPropSet = lQueryPropertySetRef.mPropertySet;
        for(uint32_t i = 0; i < lPropSet->GetPropertyInfoCount(); ++i)
        {
            const TPropertyInfo* lPropInfo = lPropSet->GetPropertyInfoAt(i);

            switch(lPropInfo->mType)
            {
            case EPropertyType::ePT_Integer:
            {
                int32_t* lPropertyDataPtr = lPropSet->GetPropertyDataPtrAt<int32_t>(i);
                lEdited = lEdited || ImGui::SliderInt(lPropInfo->mDisplayName.GetStr(),
                                                      lPropertyDataPtr,
                                                      lPropInfo->mMinIntegerValue,
                                                      lPropInfo->mMaxIntegerValue);

                break;
            }
```

```
struct TMsgQueryPropertySetRef
{
    mutable sbx::CPropertySet* mPropertySet { nullptr };
};
```

Query node property set

Iterate all properties

Check the type

Handle integer property editing

Full Example: Properties panel (2 / 2)

```
            case EPropertyType::ePT_Float:
            {
                float* lPropertyDataPtr = lPropSet->GetPropertyDataPtrAt<float>(i);
                lEdited = lEdited || ImGui::SliderFloat(lPropInfo->mDisplayName.GetStr(),
                                                        lPropertyDataPtr,
                                                        lPropInfo->mMinFloatValue,
                                                        lPropInfo->mMaxFloatValue);
                break;
            }
            case EPropertyType::ePT_Color:
            {
                TColor* lColorPtr = lPropSet->GetPropertyDataPtrAt<TColor>(i);
                lEdited = lEdited || ImGui::ColorEdit3(lPropInfo->mDisplayName.GetStr(), lColorPtr->mBuffer);
                break;
            }
            default:
                break;
            }
        }

        if(lEdited && ImGui::IsMouseDown(0))
        {
            mData->mPendingEdit = true;
        }
        else if((lEdited || mData->mPendingEdit) && !ImGui::IsMouseDown(0) && !ImGui::IsMouseDragging(0))
        {
            mData->mPendingEdit = false;
            mData->mNode->GetChannel()->BroadcastMessage(TMsgNodeEdited());
        }
    }
}
```

Handle float property editing

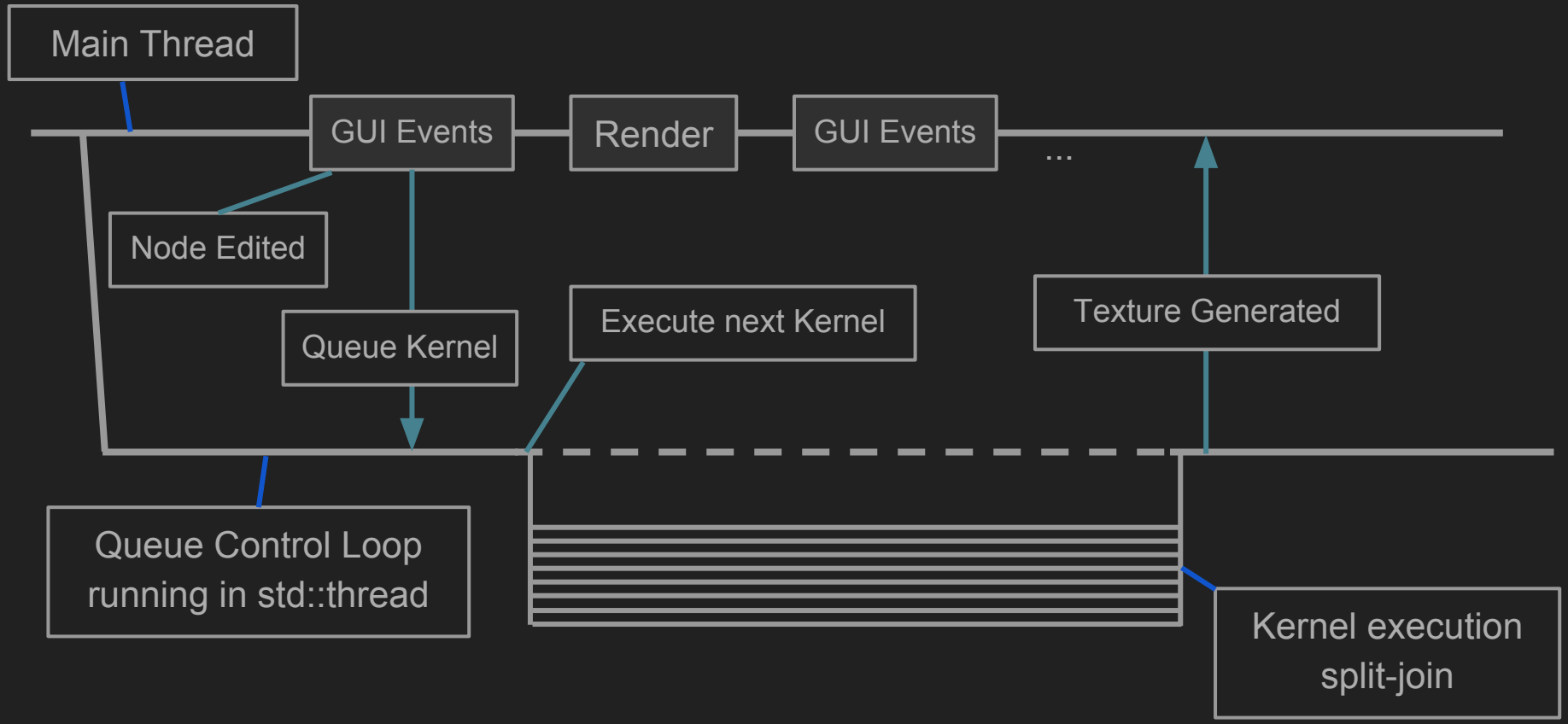
Handle color property editing

If edited, advise other interested Components

Texture Generator Module

- TGM manages the queue of kernels that generate texture data.
- Knows nothing about the node graph.
- Once Nodes are edited, they queue a kernel instance (kind of Event Queue Pattern).
- A **thread** is constantly polling the kernel queue in background.
- Kernels are executed in a thread **split-join**.
- Once finished it calls back to the Generator component.

Kernel Queue Control



Queue Control Loop with std::thread example

```
class CQueueControlLoop
{
public:
    CQueueControlLoop(TKernelQueue*    aQueue)
        : mActive(false)
        , mQueue(aQueue)
    {;}

    void Start()
    {
        mActive = true;
        mThread = std::thread([&]()
        {
            while(mActive)
            {
                std::this_thread::sleep_for( std::chrono::milliseconds(150) );
                if(!mQueue->IsEmpty())
                {
                    //pop kernel command and execute it ...
                }
            }
        });
    }
    void Stop(){ mActive = false; mThread.join(); }

private:
    std::thread      mThread;
    std::atomic_bool mActive;
    TKernelQueue*     mQueue;
};
```

construct a std::thread with a by-reference capture lambda function

Run an “infinite” loop and sleep for a while every lap

Check if the queue has something and execute it

Use std::atomic to read/write from various threads

Parallelization with OpenMP

Just one line of OpenMP *parallel for* is added to run a thread for each logic core that will generate part of the texture.

```
#pragma omp parallel for
for(int32_t y = 0; (y < mImageHeight); ++y)
{
    for(int32_t x = 0; (x < mImageWidth) && !mCancelJob; ++x)
    {
        lNextKernel->Execute(x, y);
    }
    if(mCancelJob)
    {
        //#pragma omp cancel for
        break;
    }
    lNextKernel->mFeedback->mCompletedCount += mImageWidth;
}
```

Parallelization with std::thread

```
const int32_t lMaxThreads    = _max(1u, std::thread::hardware_concurrency());
const int32_t lRowsPerThread = mImageHeight / lMaxThreads;
std::thread*  lThreads       = new std::thread[lMaxThreads];

for(int32_t t = 0; t < lMaxThreads; ++t)
{
    lThreads[t] = std::thread([&, t]()
    {
        //last thread do all the pending rows
        const int32_t lMaxRows = ((t + 1) < lMaxThreads) ? ((t + 1) * lRowsPerThread) : mImageHeight;
        for(int32_t y = t * lRowsPerThread; (y < lMaxRows) && !mCancelJob; ++y)
        {
            for(int32_t x = 0; x < mImageWidth && !mCancelJob; ++x)
            {
                lNextKernel->Execute(x, y);
                lNextKernel->mFeedback->mCompletedCount++;
            }
        }
    });
}

for(int32_t t = 0; t < lMaxThreads; ++t)
{
    lThreads[t].join();
}

delete [] lThreads;
```

Read how many logical cores have your system

Each thread computes a single consecutive chunk of the texture

Wait until all threads finished to continue processing the queue

The same can be achieved with std::thread.

Time queries with std::chrono

Use of std::chrono to get high resolution time before and after each kernel execution to know how much time it consumes.

```
krn::TKernelCommand* lNextKernel = mJobQueue.front();
mJobQueue.pop();
std::chrono::high_resolution_clock::time_point lStartTime = std::chrono::high_resolution_clock::now();
lNextKernel->mFeedback->mCompletedCount = 0;
lNextKernel->mFeedback->mExecutionTimeMilis = 0;
lNextKernel->mFeedback->mStartTime = lStartTime;
lNextKernel->mFeedback->mFinished = false;
lNextKernel->mFeedback->mRunning = true;

//[...] parallel loop

std::chrono::high_resolution_clock::time_point lEndTime = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> lTimeSpan = lEndTime - lStartTime;
lNextKernel->mFeedback->mExecutionTimeMilis = lTimeSpan.count();
```

Feedback data with std::atomic

Atomic **locks** variable memory while is being written and ensures is not coming from an **invalid cache** when read.

```
struct TKernelExecutionFeedback
{
    typedef std::chrono::high_resolution_clock::time_point TTimePoint;

    std::atomic_int32_t         mCompletedCount;
    std::atomic<double>         mExecutionTimeMilis;
    std::atomic_bool            mRunning;
    std::atomic_bool            mFinished;
    TTimePoint                  mStartTime;
    TKernelStatusHandle         mStatusHandle;

    TKernelExecutionFeedback()
        : mCompletedCount(0)
        , mExecutionTimeMilis(0)
        , mRunning(false)
        , mFinished(false)
    {
    }
};
```

Questions?

That’s All

See you in the next
BcnCppProgrammers Meetup

Thanks!

Libraries

- Dear ImGui: Immediate graphic user interface library
- GLM: OpenGL Mathematics
- stb_image & stb_image_write: read/write textures in png, tga, hdr, etc.
- nlohmann JSON: C++11 library to read and write JSON files.
- glad & KHR: OpenGL bindings.
- GLFW (not used but recommended): OpenGL/Vulkan windows.

References

Signal Template: http://simmesimme.github.io/tutorials/2015/09/20/signal-slot

Dear ImGui: https://github.com/ocornut/imgui

nlohmann json: https://github.com/nlohmann/json

Where the dead things dwell: http://www.pouet.net/prod.php?which=69691

C++ on Sea

https://cpponsea.uk/