GDC

Embedded Scripting with **Zero Overhead** in Final Builds

Arturo Cepeda
Senior Engine/Game Programmer
Deck13 Interactive

About me

- Working in the industry since 2013
- Deck13 Interactive

2014

2017

2019

(PC, PS4, Xbox One)

????

(??, ??, ??)

Embedded scripting language

- Software library
  - Compatible with a certain programming language
  - Code parser
    - Can **load** scripts written in that language
  - Code interpreter
    - Can **execute** the loaded statements
- Host application
  - Includes the library in the project
  - Binds types and functions
  - Loads/reloads scripts
  - Retrieves script values
  - Executes script functions

Embedded scripting language

- Pros
  - Hot-reloading 👍
    - Changes can be tested immediately
    - More agile workflow
  - Accessibility to Game Design and Art 👍
    - Scripts are part of the content
- Cons
  - Memory footprint 👎
    - Scripting environment
    - Loaded scripts
  - Run-time overhead 👎
    - Look-ups
    - Heap allocations
    - Garbage collection
    - Execution (interpreted code)

Scripting in the Fledge engine

- In-house engine: Fledge
- Prior to “The Surge 2”
  - Code scripting
    - Based on **Lua** (v5.1) + **tolua**
    - Only used for basic queries (e.g. “current level name?”)
- “The Surge 2”
  - New scripting system built from scratch
    - Based on **Lua** (v5.3)…
    - …and **sol2** (https://github.com/ThePhD/sol2)

Scripting in the Fledge engine

- “The Surge 2”: post-mortem
  - Scripting intensively used
    - Literally, thousands of lines of code
  - Tech dependency → Tech support 👍

Scripting in the Fledge engine

- “The Surge 2”: post-mortem
  - Dissatisfaction with Lua
    - People are rather used to C-style syntax these days
      - C#, Javascript
    - Parser not particularly helpful in many situations
      - Wasted development time because of typos

	Lua	C++
Modification of constants	*No constants*	*Compiler error*
Undeclared variable	*OK!*	*Compiler error*
Function call with unexpected type	*OK!*	*Compiler error*
Function call with wrong number of arguments	*OK!*	*Compiler error*

Scripting in the Fledge engine

- “The Surge 2”: post-mortem
  - Memory footprint
    - Critical on the consoles!
  - Run-time overhead
    - Parsing
    - Look-ups
    - Interpreted code
    - Heap allocations
    - Garbage collection

Scripting in the Fledge engine

- “The Surge 2”: post-mortem
  - Memory footprint → one only Lua state
    - Critical on the consoles!
  - Run-time overhead
    - Parsing → bytecode
    - Look-ups → cache
    - Interpreted code → “unsafe path”
    - Heap allocations → preallocated buffer + TLSF allocator
    - Garbage collection → incremental, based on steps

Scripting in the Fledge engine

- “The Surge 2”: post-mortem
  - Run-time overhead
    - Manually ported the most often called functions into C++
      - Laborious
      - Error-prone
      - The advantages of using a scripting system go away
        - Need to keep code in sync
        - Only suitable for stable code
    - Automatization considered

Fledge “Next”

- Reflection
  - Scripts no longer need to be modified in final builds
    - Scripting has only advantages as a development tool
  - Ideal situation: all scripts are part of the C++ code
    - Keep the scripting system only for development builds
    - Tool to port scripts into C++ automatically?
      - Willing to implement such a tool at first
      - Desire to replace Lua with another solution 👎

Fledge “Next”

- Reflection
  - “What about writing scripts directly in C++?”
  - “What about getting the compiler compile our scripts into native machine code?”
    - No need to even consider porting scripts into C++
    - As fast as it gets “for free”
  - “Is C++ friendly enough as a scripting language?”
    - High-level code should look familiar to people used to a C-style syntax

Fledge “Next”

- Wishes
  - Keep the good things from Lua
    - Cross-platform
    - Lightweight
    - No external dependencies
    - Free of charge
      - No budget allocated for scripting in the project
  - C++ syntax
    - Use an interpreter in development builds
      - Keep the hot-reloading feature
    - Compile the scripts into native machine code in final builds

Fledge “Next”

- Options
  - C++ interpreter (Cling)
    - A whole set of C++ standard features 👍
    - Too much for the use case 👎
      - (“Lightweight, no external dependencies”)
  - Runtime Compiled C++
    - https://github.com/RuntimeCompiledCPlusPlus
    - Open source (zlib) 👍
    - Somewhat invasive 👎
    - Only beneficial for the Tech department 👎

Fledge “Next”

- Options
  - Live++
    - https://liveplusplus.tech
    - Used in the industry (e.g. Unreal Engine) 👍
    - Suscription required for commercial projects 👎
    - Only beneficial for the Tech department 👎
  - Custom solution…?

Cflat

- Embeddable scripting language with C++ syntax
  - https://github.com/arturocepeda/cflat
  - Development tool
  - Designed with the aforementioned requirements in mind
  - Experimentation and learning
    - How to preprocess / tokenize / parse code
    - How interpreters work

Cflat

- Open source (zlib)
  - Free to use for both non-commercial and commercial projects
  - No need for credit!
- Compatible with the C++11 standard
  - Modern, yet old enough for compatibility
  - Benefiting from lambda expressions for binding functions
  - Convenience features for scripting (`auto`, range-based `for`)
- No external dependencies!
  - Only C++11 standard headers are included

Cflat

- Easy to integrate
  - One only cpp file to add to the project
- Cross-platform
  - If there is a C++11 compiler for the platform, Cflat can be used there
- Cflat ←→ C++
  - Only a (small) subset of features from C++ are available in Cflat
  - Everything from Cflat can be compiled with any C++11 compiler

Cflat

- Bindings
  - Calling methods provided by the environment
  - Convenience macros defined in Cflat

```
enum TestEnum
{
   kFirstValue,
   kSecondValue
};
```

```
Cflat::Environment env;

{
   CflatRegisterEnum(&env, TestEnum);
   CflatEnumAddValue(&env, TestEnum, kFirstValue);
   CflatEnumAddValue(&env, TestEnum, kSecondValue);
}
```

Cflat

- Bindings
  - Calling methods provided by the environment
  - Convenience macros defined in Cflat

```
struct TestStruct
{
   int var1;
   int var2;
};
```

```
{
   CflatRegisterStruct(&env, TestStruct);
   CflatStructAddMember(&env, TestStruct, int, var1);
   CflatStructAddMember(&env, TestStruct, int, var2);
}
```

Cflat

- Bindings
  - Calling methods provided by the environment
  - Convenience macros defined in Cflat

```
class TestClass
{
private:
   bool mOverload1Called;
   bool mOverload2Called;
public:
   TestClass() : mOverload1Called(false), mOverload2Called(false) {}
   void method(int pValue) { mOverload1Called = true; }
   void method(float pValue) { mOverload2Called = true; }
   bool getOverload1Called() const { return mOverload1Called; }
   bool getOverload2Called() const { return mOverload2Called; }
};
```

```
{
   CflatRegisterClass(&env, TestClass);
   CflatClassAddConstructor(&env, TestClass);
   CflatClassAddMethodVoidParams1(&env, TestClass, void, method, int);
   CflatClassAddMethodVoidParams1(&env, TestClass, void, method, float);
   CflatClassAddMethodReturn(&env, TestClass, bool, getOverload1Called);
   CflatClassAddMethodReturn(&env, TestClass, bool, getOverload2Called);
}
```

Cflat

- `CflatGlobal.h`
  - Optional helper
    - Assumes that there is a global Cflat environment
    - Requires the implementation of certain functions
  - Included in all cpp files which access scripts
  - Defines macros
    - `CFLAT_ENABLED` to distinguish between development and final
    - Can be added as a preprocessor definition in development builds

```
#if defined (CFLAT_ENABLED)
// Definitions for development builds
#else
// Definitions for final builds
#endif
```

Cflat

- `CflatGlobal.h`
  - Script inclusion

`CflatGlobalConfig.h`

```
// Relative directory where the Cflat headers are located
#define CflatHeadersPath  ./Cflat
```

Development

```
// Relative directory where the scripts are located
#define CflatScriptsPath  ./scripts
```

Final

`CflatGlobal.h`

```
#define CflatScript(pScript)  <CflatHeadersPath/Cflat.h>
```

Development

```
#define CflatScript(pScript)  <CflatScriptsPath/pScript>
```

Final

```
#include CflatScript(camera.cpp)
```

```
#include <./Cflat/Cflat.h>
```

Development

```
#include <./scripts/camera.cpp>
```

Final

Cflat

- `CflatGlobal.h`
  - Value retrieval → `CflatGet, CflatGetArray`

```
#define CflatGet(pType, pIdentifier) \
   CflatValueAs(CflatGlobal::getEnvironment()->getVariable(#pIdentifier), pType)
#define CflatGetArray(pElementType, pIdentifier) \
   CflatValueAsArray(CflatGlobal::getEnvironment()->getVariable(#pIdentifier), pElementType)
```

Development

```
#define CflatGet(pType, pIdentifier)  pIdentifier
#define CflatGetArray(pElementType, pIdentifier)  pIdentifier
```

Final

Cflat

- `CflatGlobal.h`
  - Value retrieval → `CflatGet, CflatGetArray`

Cflat (ui.cpp)

```
namespace CfUI
{
static const float kEnemyPreviewUIAlpha = 0.75f;
}
```

C++

```
#include CflatScript(ui.cpp)
...
const float uiAlpha = CflatGet(float, CfUI::kEnemyPreviewUIAlpha);
```

```
#include <./Cflat/Cflat.h>
...
const float uiAlpha =
  CflatValueAs(
    CflatGlobal::getEnvironment()->getVariable("CfUI::kEnemyPreviewUIAlpha"), float);
```

Development

```
#include <./scripts/ui.cpp>
...
const float uiAlpha = CfUI::kEnemyPreviewUIAlpha;
```

Final

Cflat

- `CflatGlobal.h`
  - Function calls → `CflatArg, CflatVoidCall, CflatReturnCall`

```
#define CflatArg(pArg)  &pArg
#define CflatVoidCall(pFunction, ...) \
   { \
      Cflat::Environment* env = CflatGlobal::getEnvironment(); \
      Cflat::Function* function = env->getFunction(#pFunction); \
      if(function) \
      { \
         env->voidFunctionCall(function, __VA_ARGS__); \
         if(env->getErrorMessage()) \
         { \
            CflatGlobal::onError(env->getErrorMessage()); \
         } \
      } \
   }
```

Development

```
#define CflatArg(pArg)  pArg
#define CflatVoidCall(pFunction, ...)  pFunction(__VA_ARGS__)
```

Final

Cflat

- `CflatGlobal.h`
  - Function calls → `CflatArg, CflatVoidCall, CflatReturnCall`

```
#define CflatReturnCall(pLValue, pReturnType, pFunction, ...) \
   { \
      Cflat::Environment* env = CflatGlobal::getEnvironment(); \
      Cflat::Function* function = env->getFunction(#pFunction); \
      if(function) \
      { \
         pLValue = \
            env->returnFunctionCall<pReturnType>(function, __VA_ARGS__); \
         if(env->getErrorMessage()) \
         { \
            CflatGlobal::onError(env->getErrorMessage()); \
         } \
      } \
   }
```

Development

```
#define CflatReturnCall(pLValue, pReturnType, pFunction, ...) \
   pLValue = pFunction(__VA_ARGS__)
```

Final

Cflat

- `CflatGlobal.h`
  - Function calls → `CflatArg, CflatVoidCall, CflatReturnCall`

Cflat (gameplay.cpp)

```
namespace CfGameplay
{
static float calculateDamage(
    Fledge::Foundation::Ref p_Target,
    Fledge::Foundation::Ref p_AbilityInstance,
    Fledge::Foundation::Ref p_AbilityHit,
    Fledge::Foundation::Ref p_Bone)
{
  ...
}
}
```

C++

```
#include CflatScript(gameplay.cpp)
...
float damage = 0.0f;
CflatReturnCall(
    damage,
    float,
    CfGameplay::calculateDamage,
    CflatArg(target),
    CflatArg(abilityInstance),
    CflatArg(abilityHit),
    CflatArg(bone));
```

Cflat

- `CflatGlobal.h`
  - Function calls → `CflatArg, CflatVoidCall, CflatReturnCall`

C++

```
#include CflatScript(gameplay.cpp)
...
float damage = 0.0f;
CflatReturnCall(
    damage,
    float,
    CfGameplay::calculateDamage,
    CflatArg(target),
    CflatArg(abilityInstance),
    CflatArg(abilityHit),
    CflatArg(bone));
```

```
#include <./Cflat/Cflat.h>
...
float damage = 0.0f;
{
   Cflat::Environment* env = CflatGlobal::getEnvironment();
   Cflat::Function* function = env->getFunction("CfGameplay::calculateDamage");
   if(function)
   {
      damage = env->returnFunctionCall<float>(function,
                                              &target,
                                              &abilityInstance,
                                              &abilityHit,
                                              &bone);
      if(env->getErrorMessage())
      {
         CflatGlobal::onError(env->getErrorMessage());
      }
   }
}
```

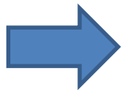
Development

Cflat

- `CflatGlobal.h`
  - Function calls → `CflatArg, CflatVoidCall, CflatReturnCall`

C++

```
#include CflatScript(gameplay.cpp)
...
float damage = 0.0f;
CflatReturnCall(
    damage,
    float,
    CfGameplay::calculateDamage,
    CflatArg(target),
    CflatArg(abilityInstance),
    CflatArg(abilityHit),
    CflatArg(bone));
```



```
#include <./scripts/gameplay.cpp>
...
float damage = 0.0f;
damage = CfGameplay::calculateDamage(
    target,
    abilityInstance,
    abilityHit),
    bone);
```

Final

Cflat

- `CflatGlobal.h`
  - Thread safety → `CflatLock, CflatUnlock`

```
#define CflatLock()  CflatGlobal::lockEnvironment();
#define CflatUnlock()  CflatGlobal::unlockEnvironment();
```

Development

```
#define CflatLock()
#define CflatUnlock()
```

Final

Cflat in the Fledge engine

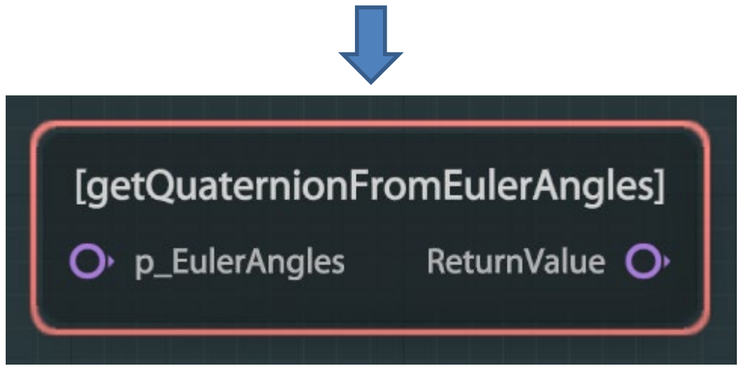
- Implementation details
  - `CflatGlobal.h` included in a common header
    - All cpps have access to the macros
  - Types and functions are bound at startup
    - C/C++ standard and engine specific → Fledge code
    - Project specific → Game code
  - Use cases
    - Access to concrete constants
    - Execution of concrete functions
    - Execution of functions by name

Cflat in the Fledge engine

- Implementation details
  - Execution of functions by name
    - Use case: custom nodes for visual scripting

Cflat

```
static Fledge::Math::Quaternion getQuaternionFromEulerAngles(
    const Fledge::Math::Vector3& p_EulerAngles)
{
  ...
}
```



Cflat

```
static float getAngleToFace(
    Fledge::Foundation::Ref p_SourceEntity,
    Fledge::Foundation::Ref p_TargetEntity)
{
  ...
}
```

Cflat in the Fledge engine

- Implementation details
  - Execution of functions by name
    - Development builds
      - Retrieve the function (`Cflat::Function`) from the environment
      - The pointer to the `Cflat::Function` instance can be cached
        - Does not change when reloading the script
      - Fill an array of arguments and call the `execute` method

Cflat in the Fledge engine

- Implementation details
  - Execution of functions by name
    - Final builds
      - Defined a fixed function signature based on the `Variant` type...

```
typedef Fledge::Foundation::Variant (*ScriptNodeFunction)(
        const Fledge::Foundation::Array<Fledge::Foundation::Variant>&);
```

(`Fledge::Foundation::Array` → `std::vector` with a custom allocator)

      - ...and a global dictionary storing the function pointers for each name

```
typedef Fledge::Foundation::Map<uint32_t, ScriptNodeFunction> ScriptNodeFunctionsMap;
static ScriptNodeFunctionsMap g_ScriptNodeFunctionsMap;
```

(`Fledge::Foundation::Map` → `std::map` with a custom allocator)

Cflat in the Fledge engine

- Implementation details
  - Execution of functions by name
    - Final builds
      - Auto-generate a wrapper for each function

Cflat

```
static float getAngleToFace(
    Fledge::Foundation::Ref p_SourceEntity,
    Fledge::Foundation::Ref p_TargetEntity)
{
  ...
}
```

Auto-generated C++

```
static Fledge::Foundation::Variant
getAngleToFace_wrapper(
    const Fledge::Foundation::Array<Fledge::Foundation::Variant>& p_Args)
{
  FLEDGE_ASSERT_FAST(p_Args.size() == 2u);
  return getAngleToFace(
      p_Args[0].as<Fledge::Foundation::Ref>(),
      p_Args[1].as<Fledge::Foundation::Ref>());
}
```

Cflat in the Fledge engine

- Implementation details
  - Execution of functions by name
    - Final builds
      - Auto-generate a wrapper for each function
      - Auto-generate wrapper registration code

Cflat

```
static float getAngleToFace(
    Fledge::Foundation::Ref p_SourceEntity,
    Fledge::Foundation::Ref p_TargetEntity)
{
  ...
}
```

Auto-generated C++

```
static Fledge::Foundation::Variant
getAngleToFace_wrapper(
    const Fledge::Foundation::Array<Fledge::Foundation::Variant>& p_Args)
{
  FLEDGE_ASSERT_FAST(p_Args.size() == 2u);
  return getAngleToFace(
      p_Args[0].as<Fledge::Foundation::Ref>(),
      p_Args[1].as<Fledge::Foundation::Ref>());
}
```

```
...
g_ScriptNodeFunctionsMap[FledgeCompileTimeHash32(getAngleToFace)] = getAngleToFace_wrapper;
...
```

Cflat in the Fledge engine

- Development environment
  - Using an external code editor
    - Internal documentation
    - Convenience features
    - Work does not get lost on editor crash
  - Automatic hot-reload on script save
    - Info log on success
    - Error log if there are any errors
      - In that case, the old version remains loaded

Cflat in the Fledge engine

- Development environment
  - Header files included from the scripts
    - Declarations for development builds (internal documentation)
    - The required `#include` directives for final builds
  - `_fledge.h`
    - C/C++ standard functions
    - Engine specific types
  - `_game.h`
    - Project specific types

```
#if defined (CFLAT_ENABLED)

// Declarations, used as internal documentation

#else

// #include directives for final builds

#endif
```

Cflat in the Fledge engine

- Development environment
  - Visual Studio Code
    - https://code.visualstudio.com/
    - Open source generic code editor
    - C++ extension
      - Syntax highlighting
      - Error highlighting
      - Auto-completion
      - Auto-formatting (`clang-format`)
    - Settings can be saved into a JSON file and checked-in
      - Scripting users don’t need to configure anything on their side

Cflat in the Fledge engine

- Development environment
  - Visual Studio Code

```
struct Camera
{
  /// Returns the active camera component
  static Fledge::Foundation::Ref getActiveCamera();

  /// Returns a reference to the owner entity
  static Fledge::Foundation::Ref getEntity(Fledge::Foundation::Ref p_Ref);

  /// Returns the screen position for the given world position
  static Fledge::Math::Vector2 getScreenPosition(
      Fledge::Foundation::Ref p_Ref,
      const Fledge::Math::Vector3& p_WorldPosition);
};
```

Cflat in the Fledge engine

- Development environment
  - Visual Studio Code

```
298  //-----------------------------------------------------------------------------
299  static Fledge::Math::UVector2
300  worldToScreen(const Fledge::Math::Vector3& p_WorldPosition)
301  {
302    using Fledge::Core::Component::Camera;
303
304    Fledge::Foundation::Ref camera = Camera::getActiveCamera();
305    const Fledge::Math::Vector2 screenPosition = Camera::
306  }
307  //-----------------------------------------------------------------------------
308
309
```

```
getActiveCamera
getEntity
getScreenPos…   static Fledge::Math::Vector2 Fledg…
```

```
struct Camera
{
  /// Returns the active camera component
  static Fledge::Foundation::Ref getActiveCamera();

  /// Returns a reference to the owner entity
  static Fledge::Foundation::Ref getEntity(Fledge::Foundation::Ref p_Ref);

  /// Returns the screen position for the given world position
  static Fledge::Math::Vector2 getScreenPosition(
      Fledge::Foundation::Ref p_Ref,
      const Fledge::Math::Vector3& p_WorldPosition);
};
```

Cflat in the Fledge engine

- Development environment
  - Visual Studio Code

```
struct Camera
{
  /// Returns the active camera component
  static Fledge::Foundation::Ref getActiveCamera();

  /// Returns a reference to the owner entity
  static Fledge::Foundation::Ref getEntity(Fledge::Foundation::Ref p_Ref);

  /// Returns the screen position for the given world position
  static Fledge::Math::Vector2 getScreenPosition(
      Fledge::Foundation::Ref p_Ref,
      const Fledge::Math::Vector3& p_WorldPosition);
};
```

```
298  //-------------------------------------------------------------------------
299  static Fledge::Math::UVector2
300  worldToScreen(const Fledge::Math::Vector3& p_WorldPosition)
301  {
302    using Fledge::Core::Component::Camera;
303
304    Fledge::Foundation::Ref camera = Camera::getActiveCamera();
305    const Fledge::Math::Vector2 screenPosition = Camera::
306  }
307  //-------------------------------------------------------------------------
308
309
```

```
getActiveCamera
getEntity
getScreenPos…   static Fledge::Math::Vector2 Fledg…
```

```
298  //-------------------------------------------------------------------------
299  static Fledge::Math::UVector2
300  worldToScreen(const Fledge::Math::Vector3& p_WorldPosition)
301  {
302    using Fledge::Core::Component::Camera;
303
304    Fledge::Foundation::Ref camera = Camera::getActiveCamera();
305    const Fledge::Math::Vector2 screenPosition = Camera::getScreenPosition()
306  }
307  //-------------------------------------------------------------------------
```

```
Fledge::Math::Vector2
getScreenPosition(Fledge::Foundation::Ref p_Ref, const
Fledge::Math::Vector3 &p_WorldPosition)

Returns the screen position for the given world position
```

Cflat in the Fledge engine

- Workflow for Tech
  - Exposing types:
    - Bind code in the engine or game
    - Trigger a new editor build
    - Extend the corresponding header file for scripting (`_fledge.h`/`_game.h`)
      - Add the declarations in the `CFLAT_ENABLED` block (development)
      - Add the `#include` directives in the other block (final)

Cflat in the Fledge engine

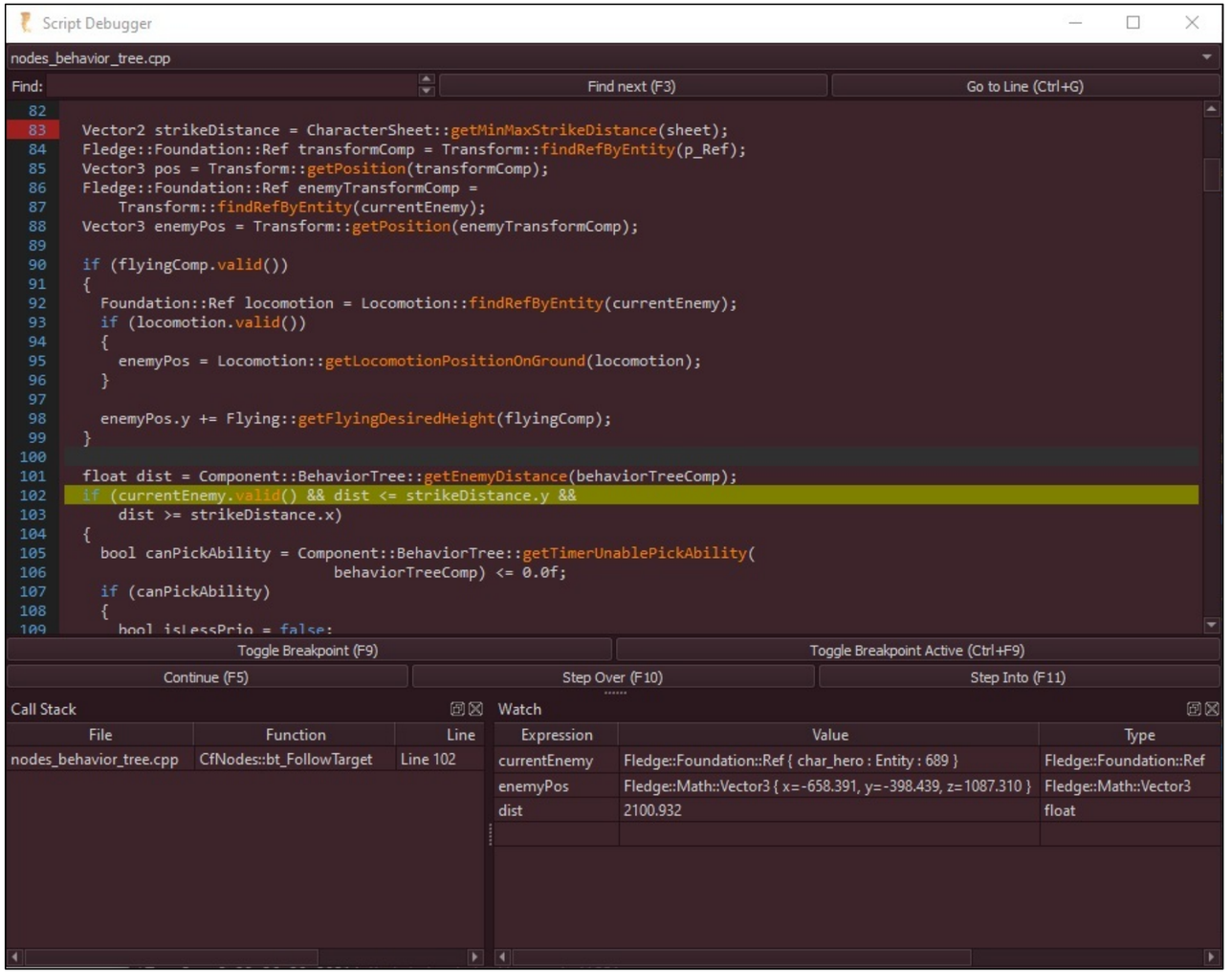
- Debugging tools
  - Logs
    - Macros defined on the engine side, and bound for scripting
    - `LOG_DEBUG`, `LOG_INFO`, `LOG_WARNING`, `LOG_ERROR`
    - Logs show up on the editor's console

Cflat

```
LOG_DEBUG << "Hit! -- Target: '"
          << target.name()
          << "' -- Damage: "
          << damage
          << LOG_FLUSH;
```

Cflat in the Fledge engine

- Debugging tools
  - Script debugger
    - Code viewer
    - Breakpoint support
    - Step over/into
    - “Call Stack” view
    - “Watch” view
      - Value inspection
      - Value manipulation



Cflat in the Fledge engine

- Debugging tools
  - Script debugger
    - Cflat provides an execution hook

```
struct CallStackEntry
{
   const Program* mProgram;
   const Function* mFunction;
   uint16_t mLine;

   CallStackEntry(const Program* pProgram, const Function* pFunction = nullptr);
};

typedef CflatSTLVector(CallStackEntry) CallStack;
```

```
typedef void (*ExecutionHook)(Environment* pEnvironment, const CallStack& pCallStack);
```

Cflat in the Fledge engine

- Debugging tools
  - Script debugger
    - The editor is a standalone application
    - It communicates with the game through the network
    - Network packets are handled in a thread
    - Script execution paused and resumed using a semaphore

Considerations

- The Cflat parser is **not** the C++ compiler
  - “Cross-platform” situation
    - The code can compile on one platform…
    - ...which doesn’t mean it also compiles for the rest (VC++ vs. Clang)
    - Ultimate goal: not allowing anything a C++11 compiler would not allow
  - `#include` directives
    - Ignored in Cflat…
      - Everything loaded in the environment is accessible
      - Only the order in which the scripts are loaded needs to be kept in mind
    - …but still required in C++!

Considerations

- Scripts are included as headers in final builds
  - Beware of non static functions
    - Including a script in more than one cpp would be a linking error
  - Beware of `using namespace` statements
    - Avoid them in the global scope!
    - If used inside a block, previous included headers still matter
      - The rest of the cpp is safe…
      - …but there might be errors in the script code itself
- Necessity of checking compilation in final builds
  - Script as a separate cpp
  - Client cpp that includes the script

Thank you!

Email: **acepeda@deck13.com**
Twitter: **@acepedaperez**

EXCITED ABOUT WHAT'S NEXT? JOIN US!

We already have a lot of new things on our lists…

Sarah Wrensch

Human Resources Manager

jobs@deck13.com

www.deck13.com