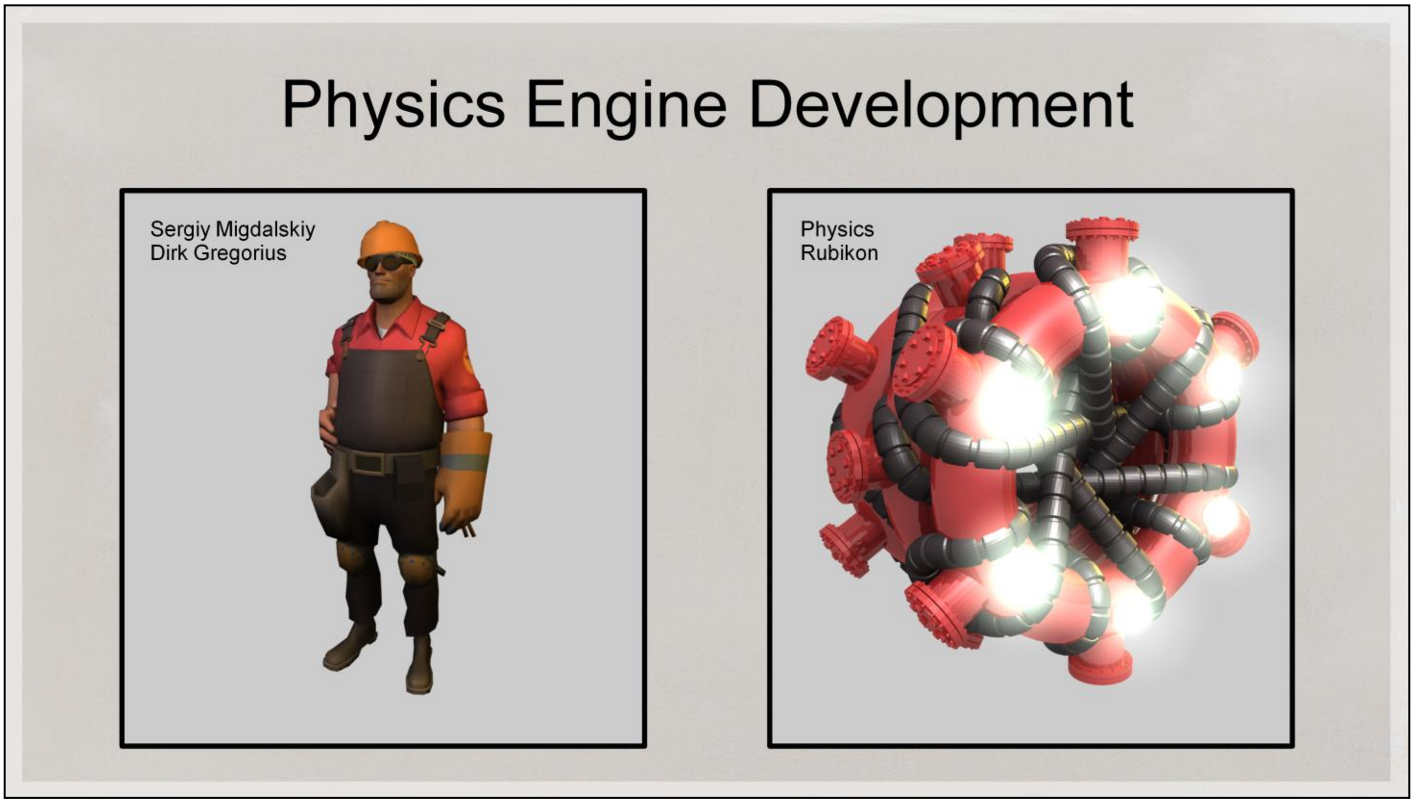


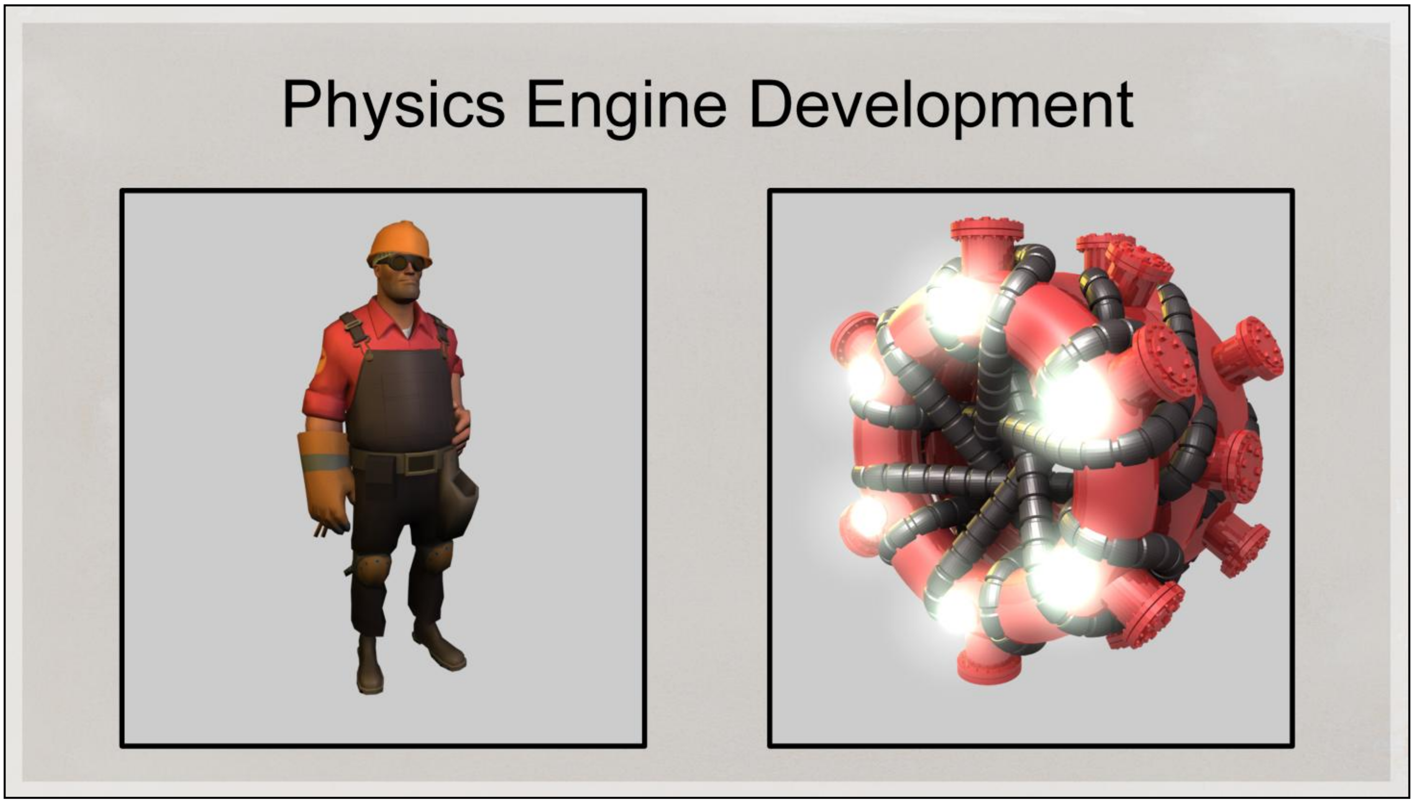
Physics Engine Development

Sergiy Migdalskiy
Dirk Gregorius

Physics
Rubikon

Hello!

My name is Sergiy Migdalskiy and I work at Valve with Dirk Gregorius. I helped **port**, develop and optimize our in-house physics engine, Rubikon. Also, I have implemented a physics engine for **Uncharted**: Drake's Fortune on PS/3.



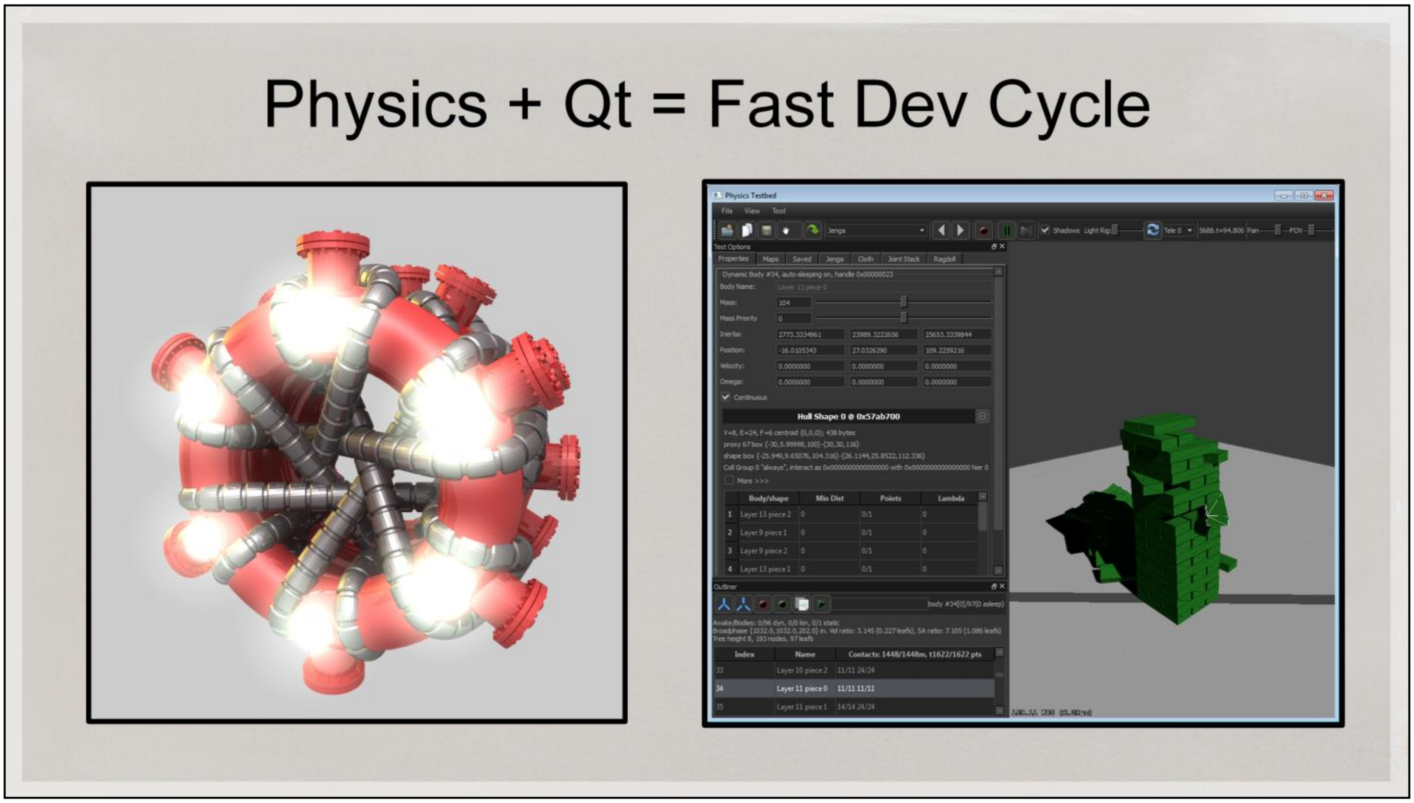
Physics Engine Development

While this talk is about physics development, I'd like to emphacise that tools and techniques I'm describing are general purpose. They are probably applicable to any complex system written in C++.

Visualization

Run Playing_with_Physics.mov

This is what I do the whole day long.
I play with physics, throw ragdolls into sandboxes, and such silliness.
(wait) Here I'm piling up some ragdolls, it's looking good, until..
I notice one of the ragdolls, the very first, doesn't seem to collide with anything.
I'm testing some more. I make sure the other ragdolls work fine. Then I notice something.
Here it is – the collision flags on the very first ragdoll are set to not collide with anything.
Now the mystery is solved, and I go on to other tests.



Physics + Qt = Fast Dev Cycle

5688.t=94.806

Dynamic Body #34, auto-sleeping on, handle 0x00000023

Body Name:	Layer 11 piece 0		
Mass:	104		
Mass Priority	0		
Inertia:	2773.3334961	23989.3222656	25653.3339844
Position:	-16.0105343	27.0326290	109.2259216
Velocity:	0.0000000	0.0000000	0.0000000
Omega:	0.0000000	0.0000000	0.0000000

☑ Continuous

Hull Shape 0 @ 0x57ab700

V=8, E=24, F=6 centroid {0,0,0}; 438 bytes
proxy 67 box {-30,5.99998,100}-{30,30,116}
shape box {-25.949,9.65076,104.316}-{26.1144,25.8522,112.336}
Coll Group 0 "always", interact as 0x0000000000000000 with 0x0000000000000000 hier 0

□ More >>>

	Body/shape	Min Dist	Points	Lambda
1	Layer 13 piece 2	0	0/1	0
2	Layer 9 piece 1	0	0/1	0
3	Layer 9 piece 2	0	0/1	0
4	Layer 13 piece 1	0	0/1	0

body #34[0]/97(0 asleep)

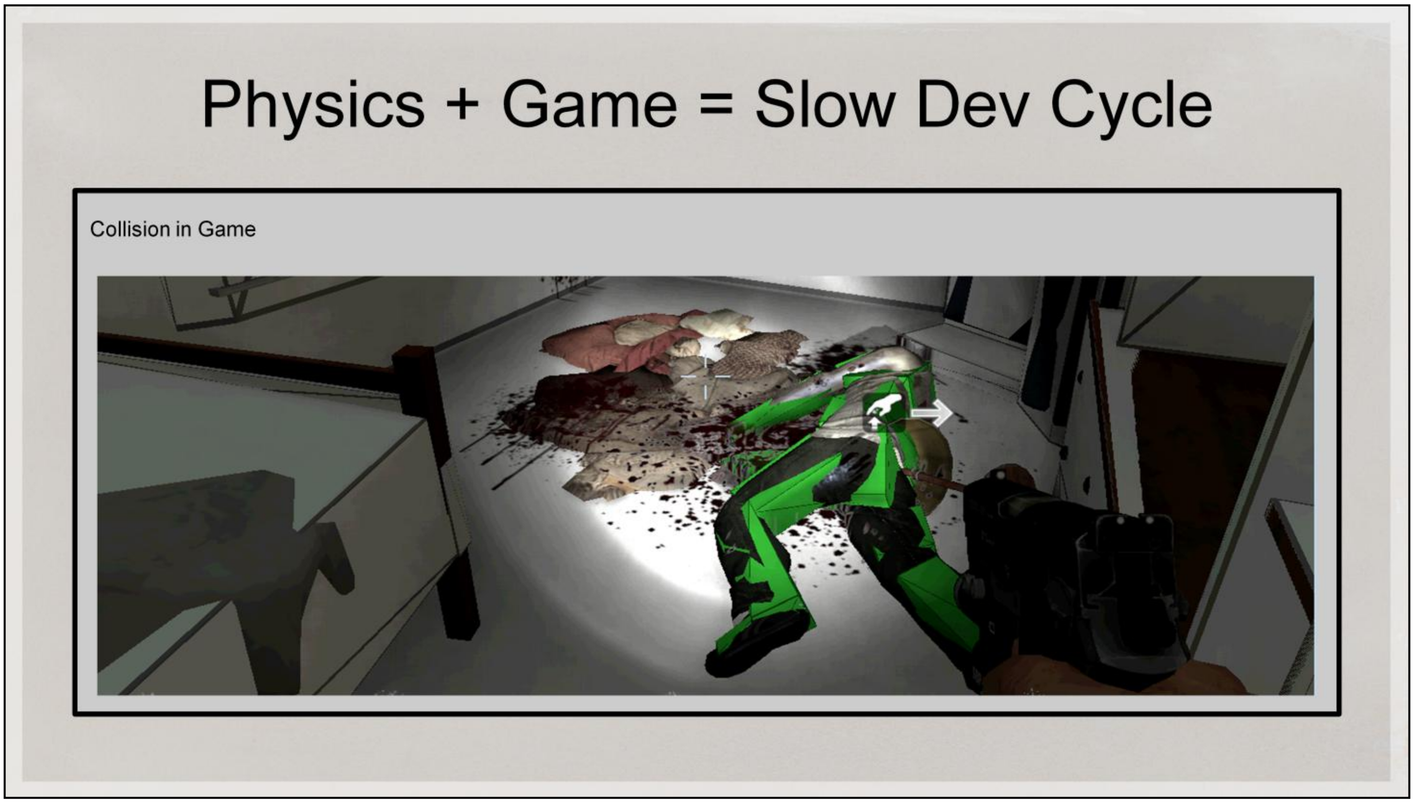
Awake/Bodies: 0/96 dyn, 0/0 kin, 0/1 static
Broadphase {1032.0,1032.0,202.0} in. Vol ratio: 3.145 (0.227 leafs), SA ratio: 7.105 (1.086 leafs)
Tree height 8, 193 nodes, 97 leafs

Index	Name	Contacts: 1448/1448m, t1622/1622 pts
33	Layer 10 piece 2	11/11 24/24
34	Layer 11 piece 0	11/11 11/11
35	Layer 11 piece 1	14/14 24/24

We started working with Dirk by porting his rigid body engine to Valve's **framework**. I knew Qt and I like playing with stuff I'm **working on**, for fun and to make it **easier**. So I implemented a little IDE for our physics **project**.

It was incremental. Every time something took more than 30 seconds seconds to look up in the debugger, I made a widget that makes it a split-second.

We had physics running in an isolated tool with buttons, sliders and 3D window with **shadows**, the one you've just seen, which was pretty **cool**. And it restarted in **seconds**. We call it the Physics Testbed, or the Physics Debugger. Not to be confused with Visual Studio debugger.



Physics + Game = Slow Dev Cycle

Collision in Game

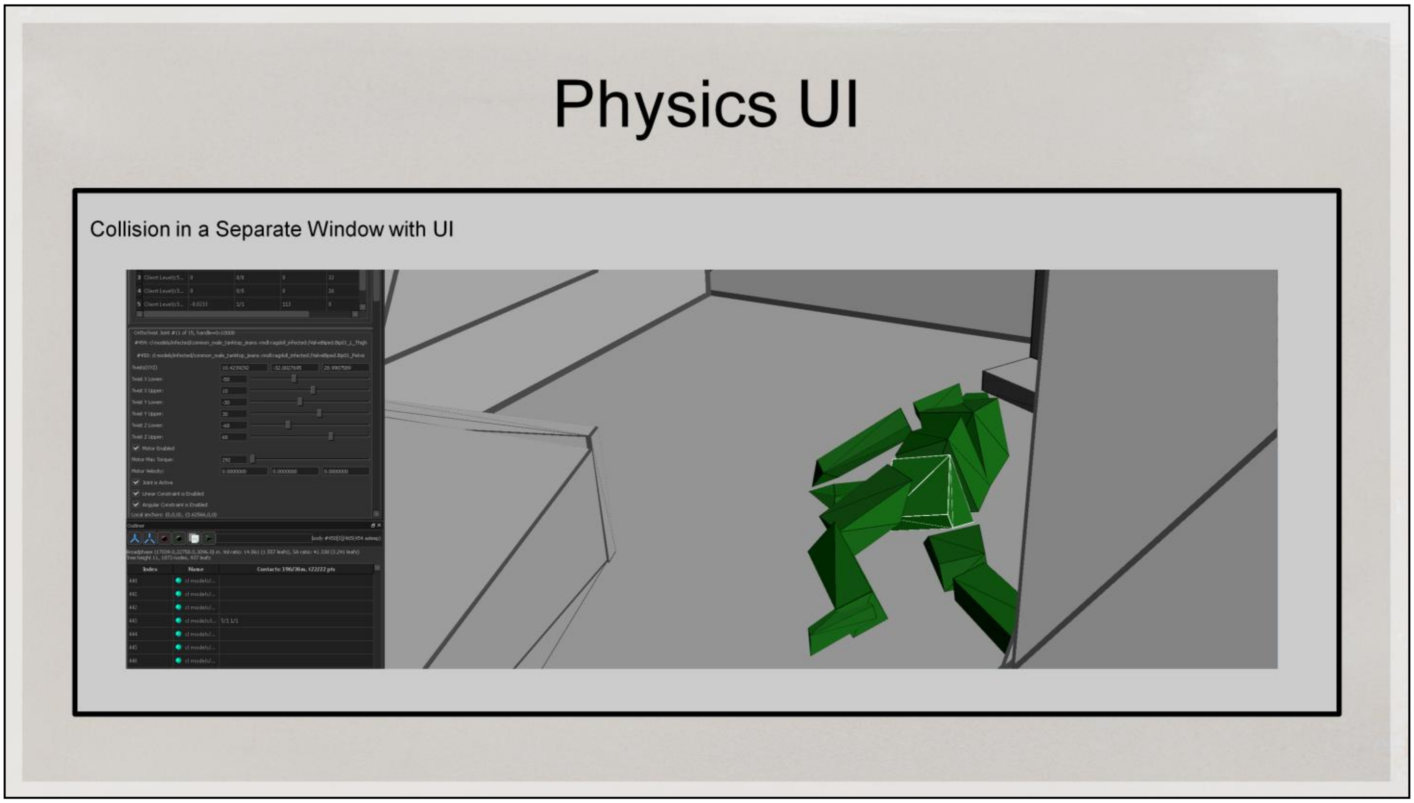
But there comes the time for every game physics engine to be put into an actual game.

Once we started integrating physics with the game, we had to debug it inside the game. So we lost our fast-iteration, nice UI.

Every engine has in-game visualization of physics. So we make do.

It has some serious drawbacks, though.

- You can't easily use mouse cursor for clicking collision or UI.
- You cannot fly around easily, you have to implement a special camera for it.
- You need to render inside game engine, which can be pretty tricky.
- You have to implement a special pause that pauses absolutely everything while you're examining something.
- It's non-trivial to implement rich UI, like you can do with Qt.



Physics UI

Collision in a Separate Window with UI

3	Client Level(c5...	0	0/0	0	32
4	Client Level(c5...	0	0/0	0	16
5	Client Level(c5...	-0.0233	1/1	113	0

OrthoTwist Joint #11 of 15, handle=0x10008

#454: cl models/infected/common_male_tanktop_jeans.vmdl:ragdoll_infected:/ValveBiped.Bip01_L_Thigh

#450: cl models/infected/common_male_tanktop_jeans.vmdl:ragdoll_infected:/ValveBiped.Bip01_Pelvis

Twists(XYZ) 10.4239292 -32.0027695 28.9907589

Twist X Lower: -50

Twist X Upper: 10

Twist Y Lower: -30

Twist Y Upper: 30

Twist Z Lower: -68

Twist Z Upper: 68

☑ Motor Enabled

Motor Max Torque: 292

Motor Velocity: 0.0000000 0.0000000 0.0000000

☑ Joint is Active

☑ Linear Constraint is Enabled

☑ Angular Constraint is Enabled

Local anchors: {0,0,0}, {3.62566,0,0}

Outliner

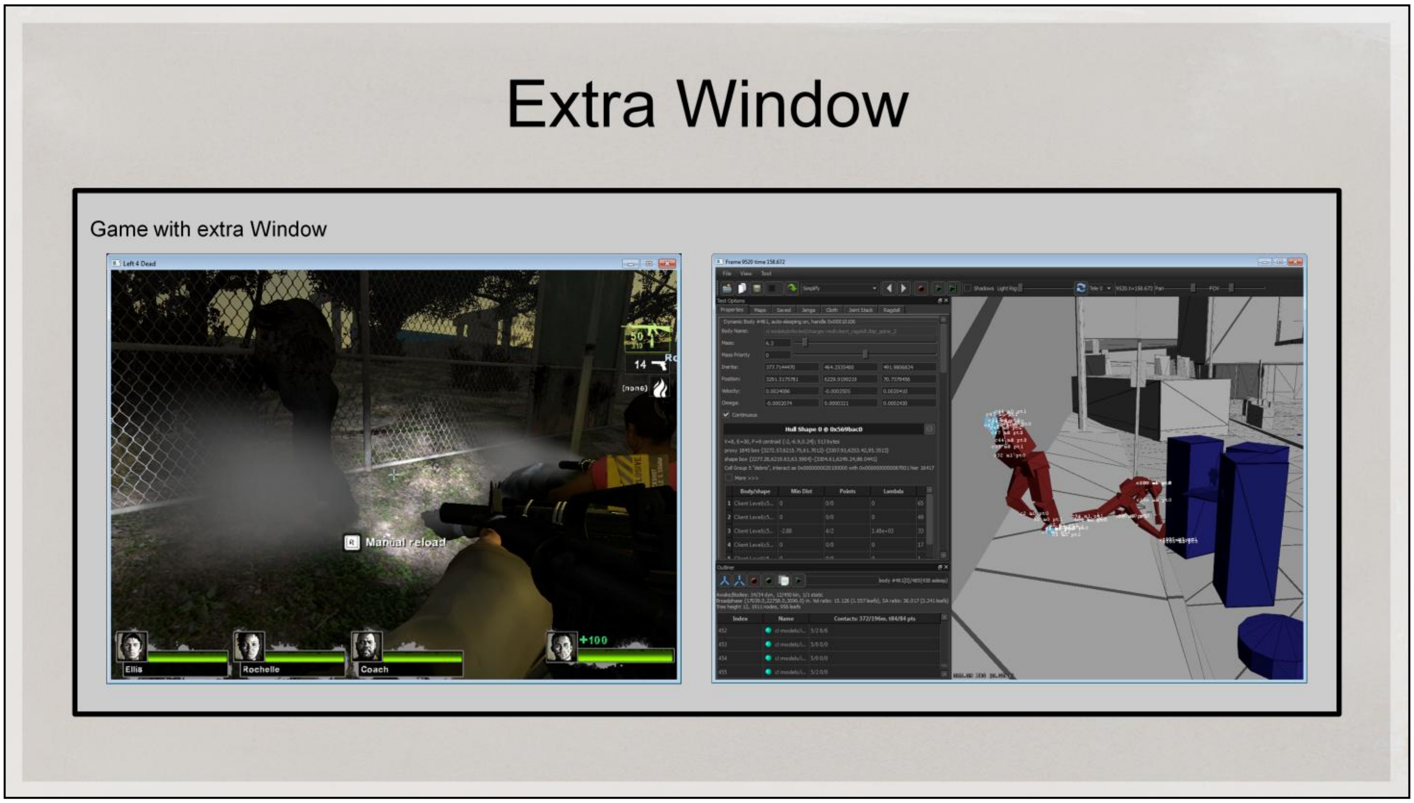
body #450[0](465(454 asleep)

Broadphase {17039.0,22758.0,3096.0} in. Vol ratio: 14.0x) (1.557 leafs), SA ratio: 41.338 (3.24) leafs)
Tree height 11, 1873 nodes, 937 leafs

Index	Name	Contacts: 190/30m, 122/22 pts
440	cl models/...	
441	cl models/...	
442	cl models/...	
443	cl models/...	5/1 1/1
444	cl models/...	
445	cl models/...	
446	cl models/...	

Physics Testbed was a great UI with a bunch physics-centric widgets. It was a great help already in making physics work. I felt I can reuse it in the game integrated physics.

But how do we display in-game physics world in a separate window?



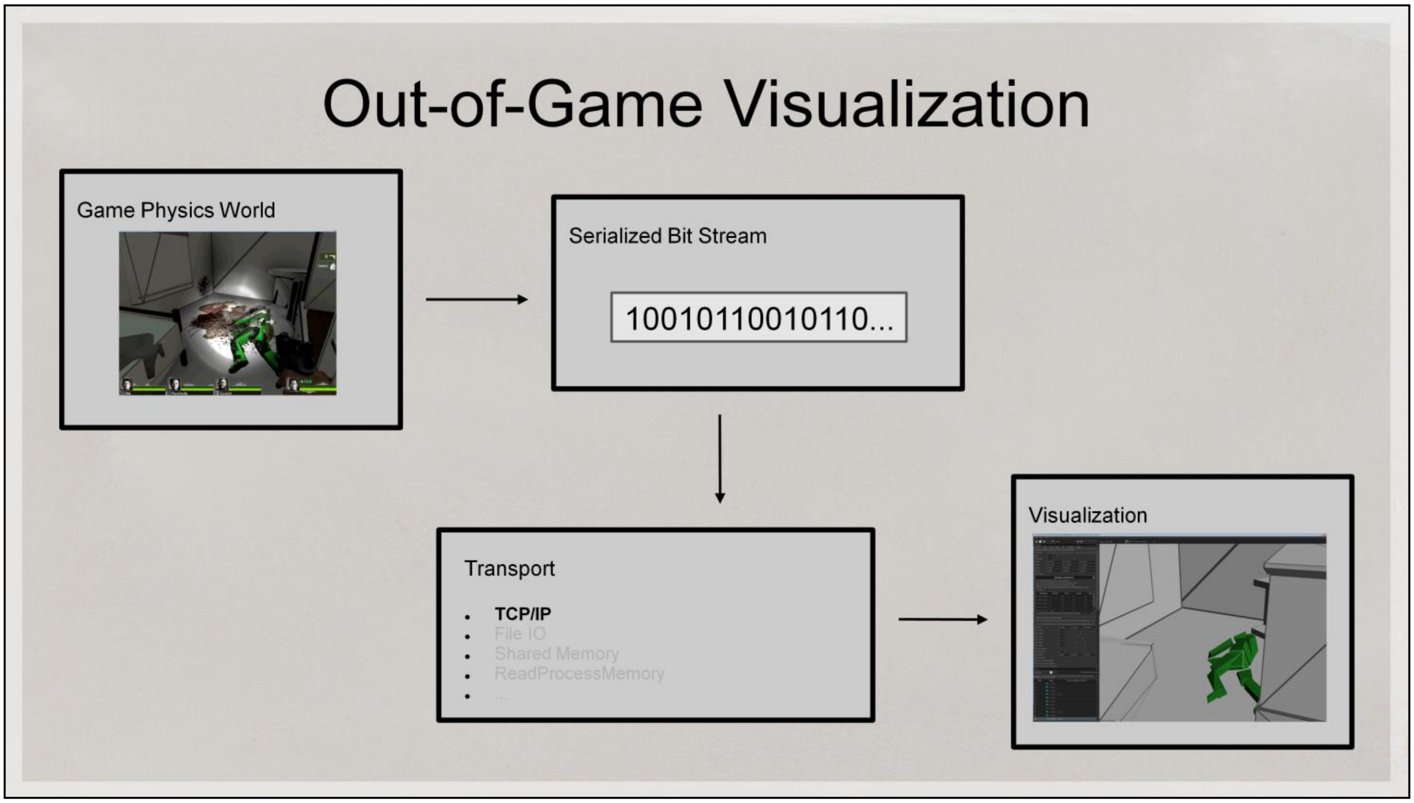
Extra Window

Game with extra Window

I briefly entertained an idea to open an extra Qt window in the game process. I find it:

- inconvenient to implement
- It's intrusive to the main game loop
- does not work when the game is crashed or stopped on a breakpoint
- You have to implement “full pause” in the game, both on server and client

So I decided against in-process window. This means I could just reuse the existing separate tool, our Physics Testbed.



Out-of-Game Visualization

Game Physics World

Serialized Bit Stream

10010110010110...

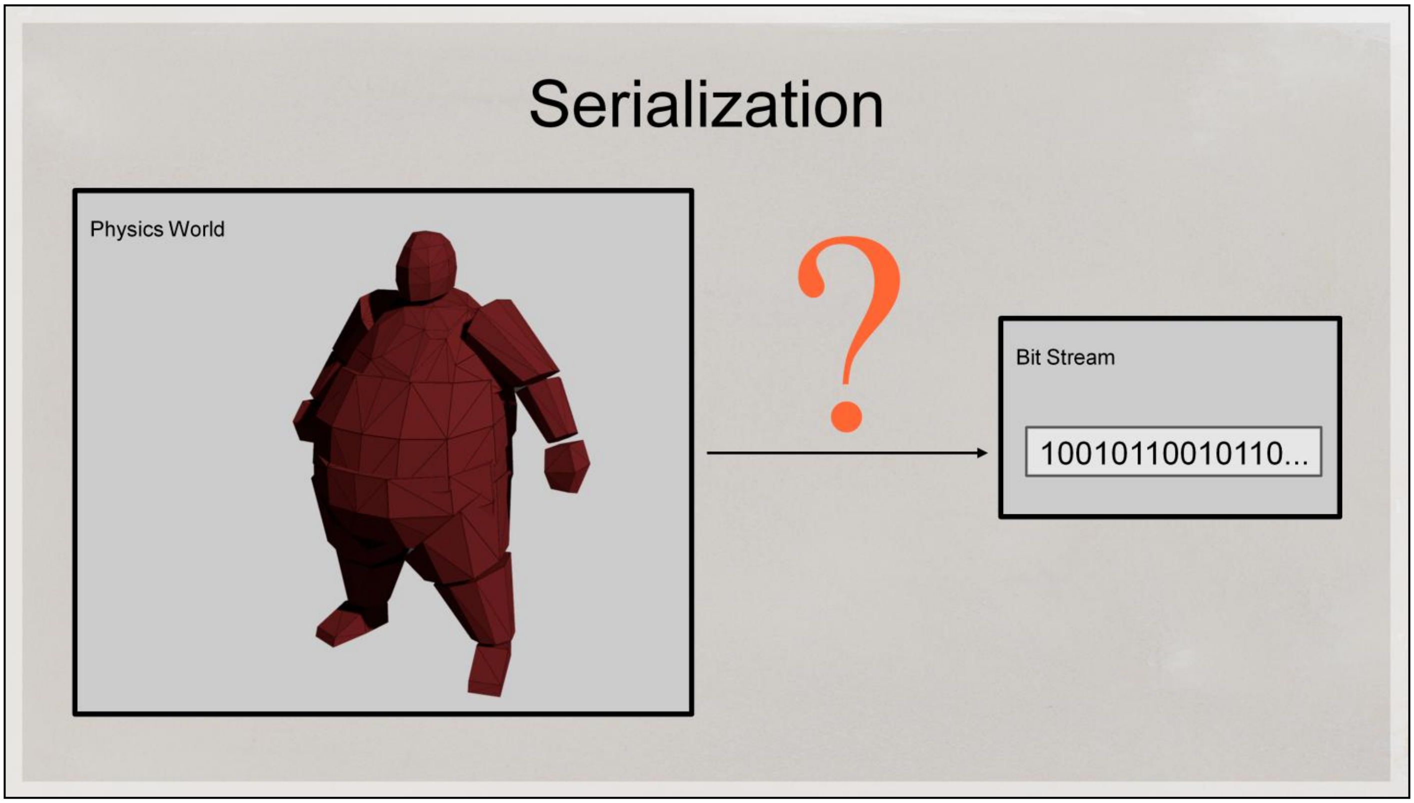
Transport

- **TCP/IP**
- File IO
- Shared Memory
- ReadProcessMemory
- ...

Visualization

To visualize in external process, you need to send data to it.
The hardest part about sending physics data to external app is serialization.
But if you can properly and fully serialize your world state into a stream of bits and send it over the wire, it becomes very straightforward to visualize your physics in a separate app.

Playing with physics in external app is a nice bonus feature.



Serialization

Physics World

?

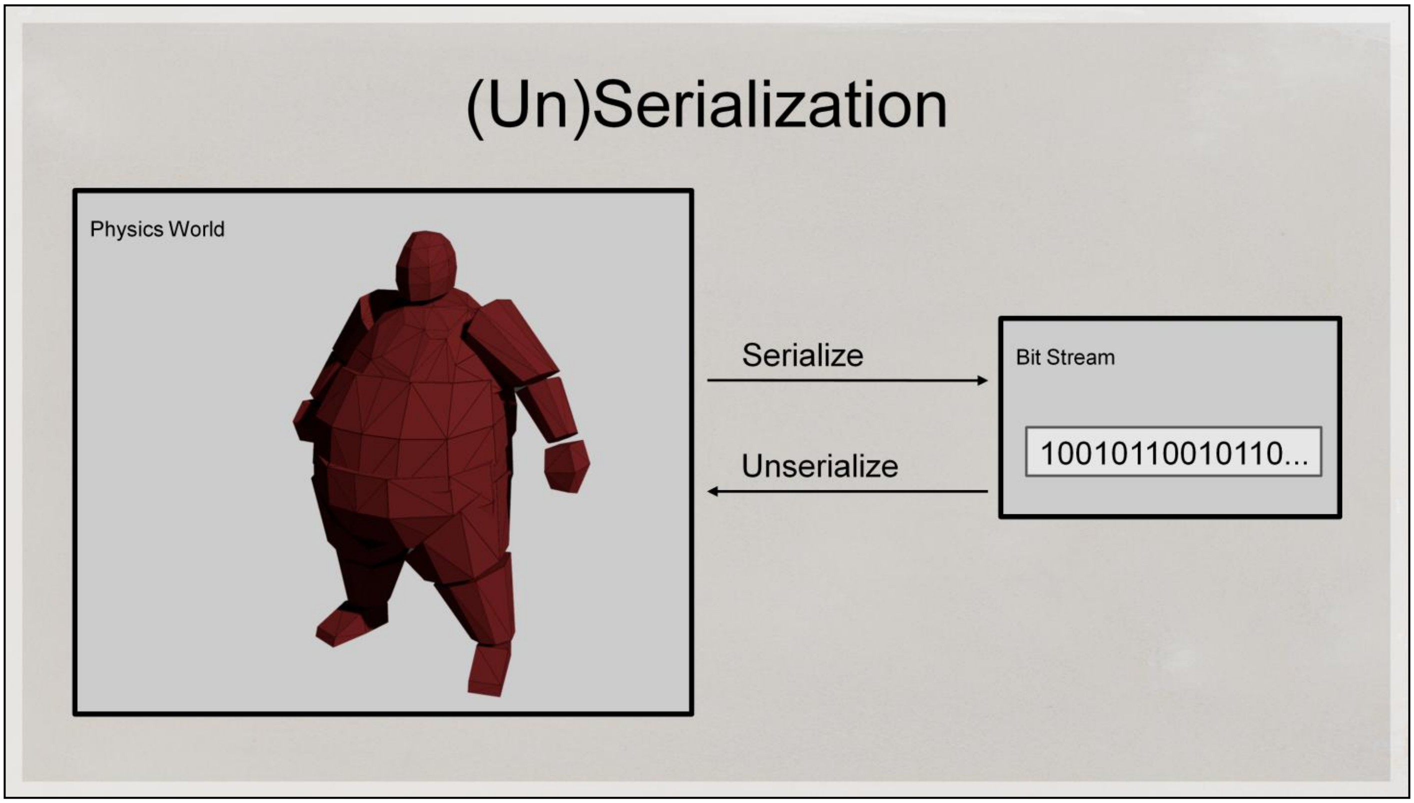
Bit Stream

10010110010110...

So serialization is the biggest obstacle here. But once we solve it, it becomes very useful:

- we can save and look at something later
- we can implement nice data and time profilers
- we can record and replay bugs

Once you can serialize, it is trivial to stream to external app.



(Un)Serialization

Physics World

Serialize

Unserialize

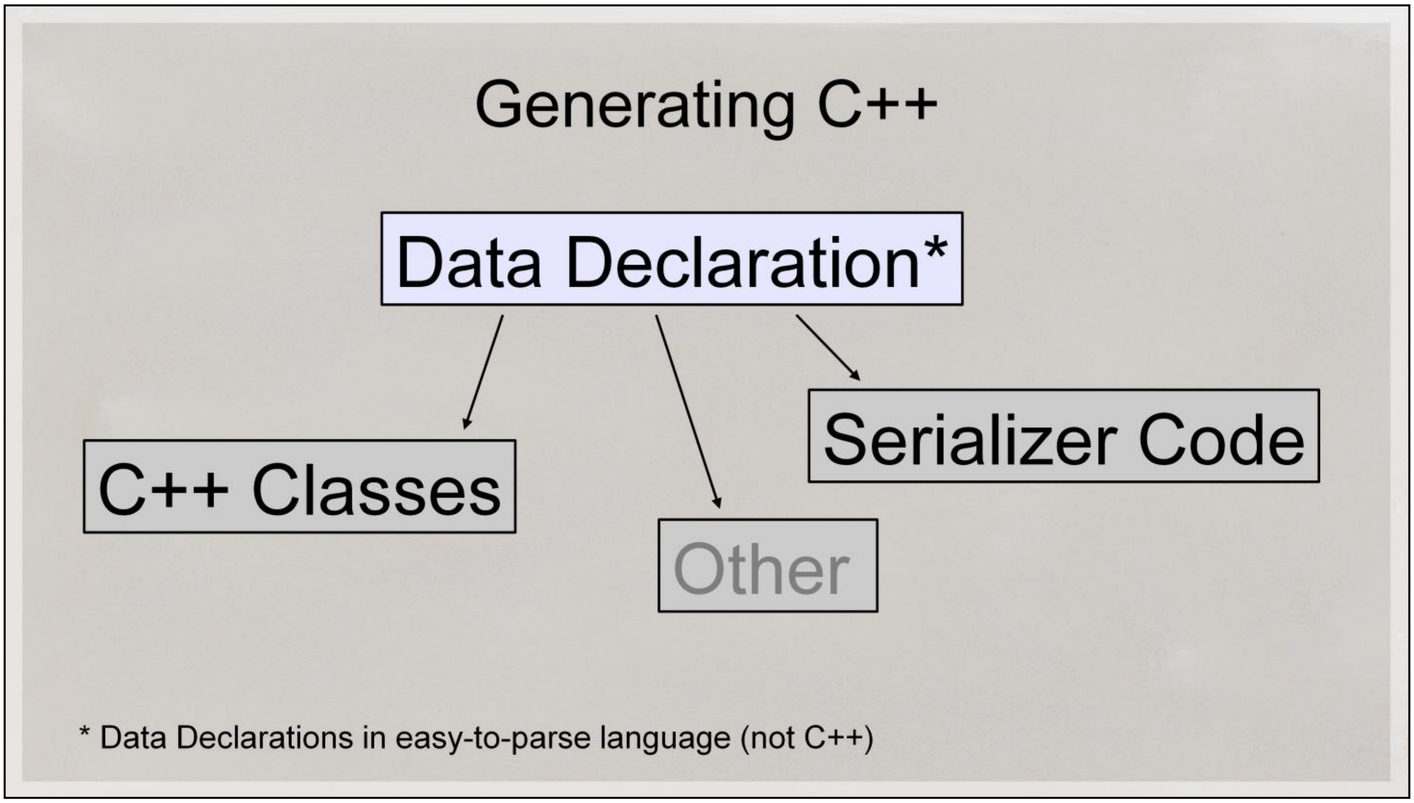
Bit Stream

10010110010110...

By the way, of course, serialization works two ways. The external app has to somehow turn our stream of bits into data structures..

So How do we serialize? Physics engine is complex. There are so many classes and data structures in there! Serializing pointers is hard. How do we build software that doesn’t miss anything, and then reconstructs everything faithfully. And hopefully it updates itself every time we add or remove a field in a class?

Definitely not! by writing serialization by hand.



Generating C++

Data Declaration*

C++ Classes

Other

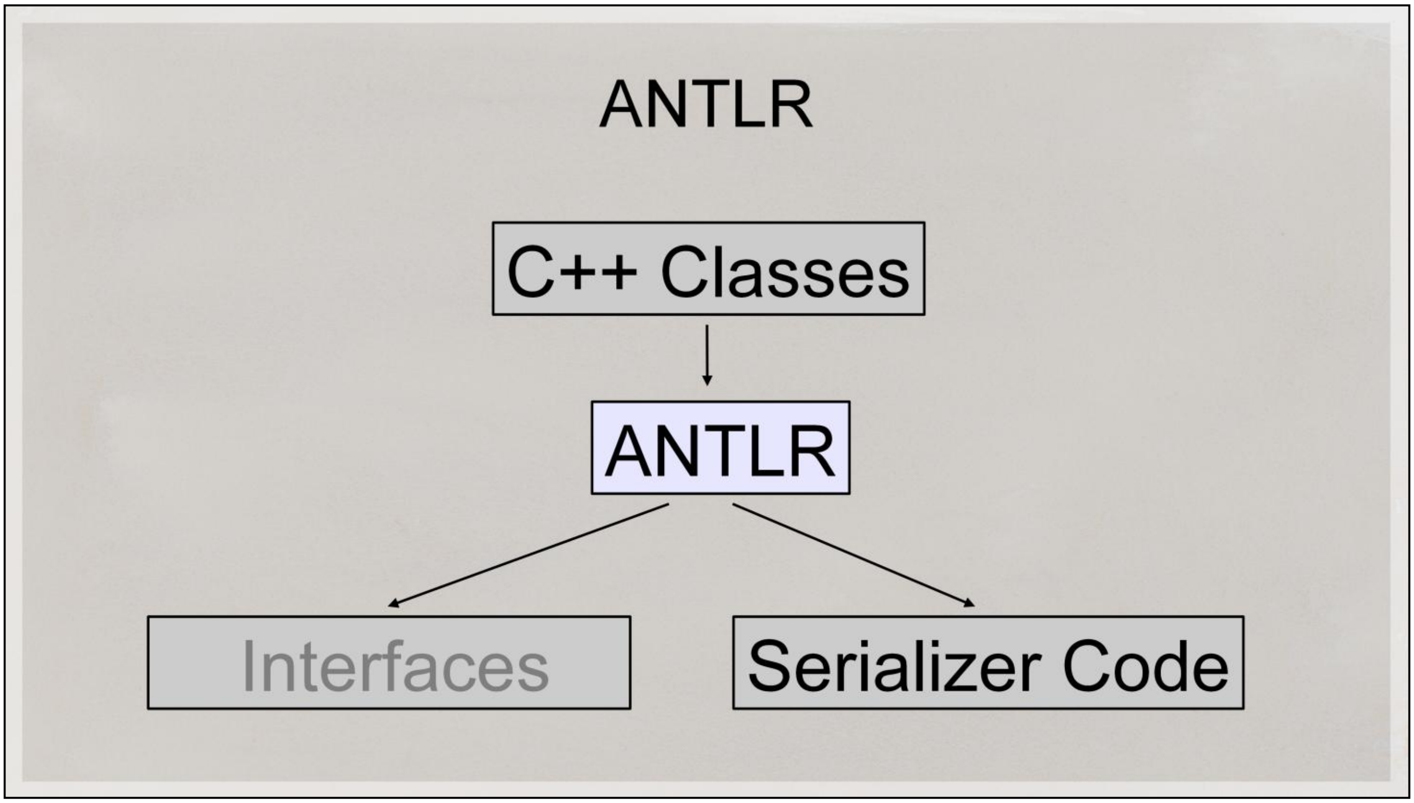
Serializer Code

* Data Declarations in easy-to-parse language (not C++)

- One way to solve it is to describe your data structure in non-C++ (like protobufs), and generate C++ code from that.
- The description has to be easy to parse

We already had C++ code to start with, and the data structures were not easy to express in something like protobufs. I'd have to struggle to define the language and then struggle to express our nontrivial C++ data structures.

If only we use C++! Then there is no need to remember the new language syntax. Anybody can come in and modify the code without learning an extra language



ANTLR

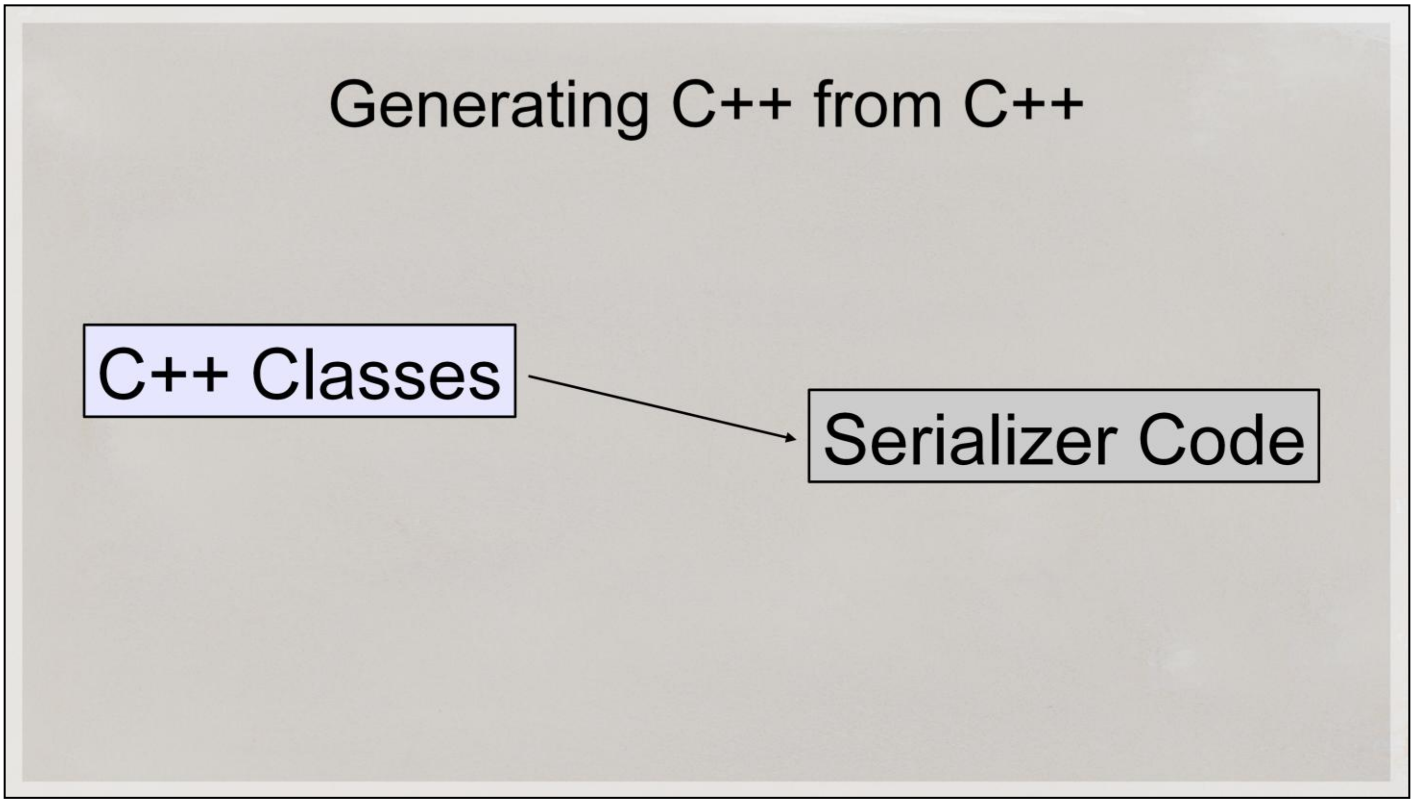
C++ Classes

ANTLR

Interfaces

Serializer Code

- ANTLR is a parser generator and was the first thing I used. It has a whole IDE for debugging grammars. It's very powerful.
- In house, we use ANTLR for our reflection API. We only parse a subset of C++ specification. I took that parser and I still use it to generate clean public interfaces to our physics engine. They take care of type conversions, they log external activity and do some other things.
- ANTLR is not ideal for parsing C++, though. You have to maintain C++ grammar, Java is its native language so it's the easiest target to generate parsers. It's fast, but an optimized C++ parser is faster.

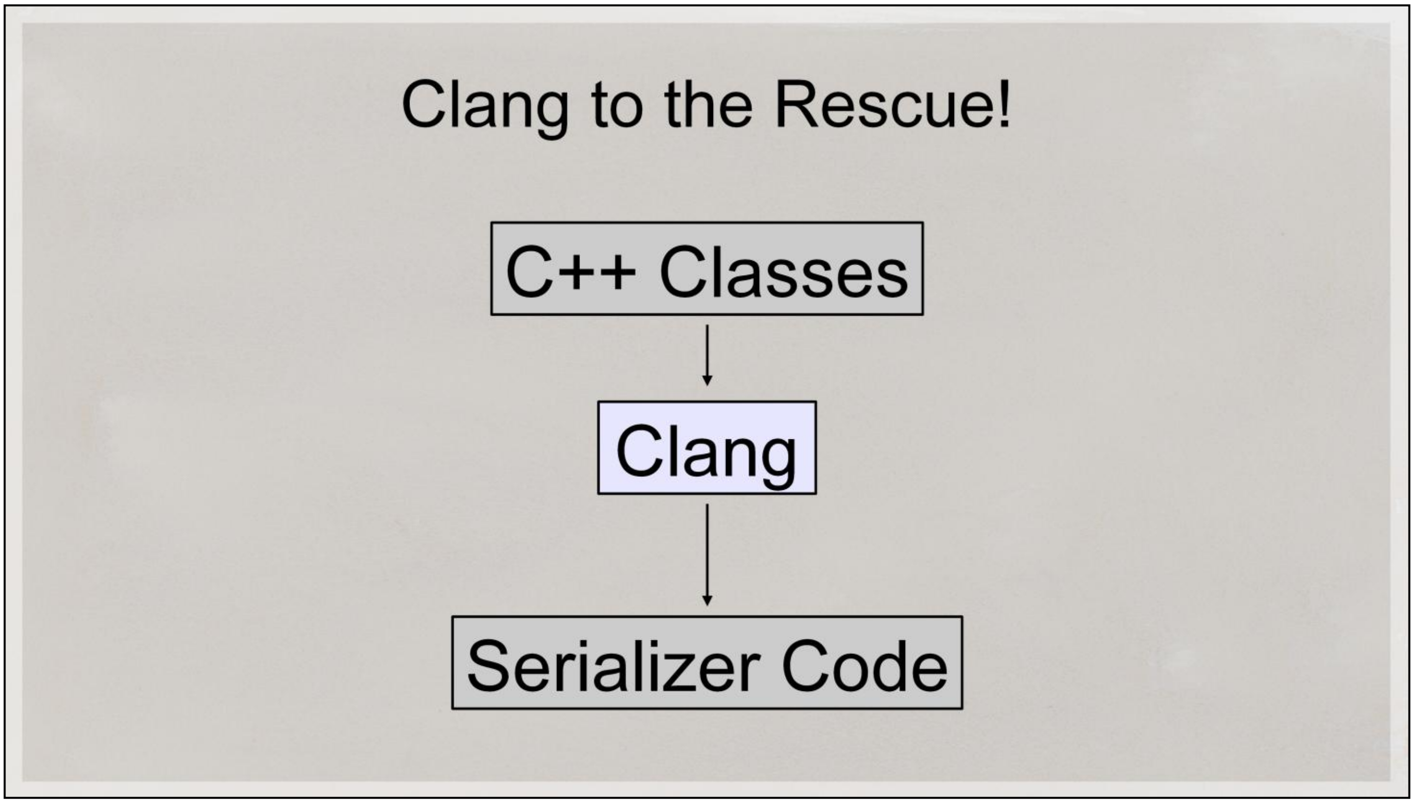


Generating C++ from C++

C++ Classes

Serializer Code

We can analyze the engine's header files to generate serialization code.
That's what we would do if we wrote the serialization by hand.
But instead of doing it by hand we can parse C++ in a utility and spit out C++ which will serialize our data.
But parsing C++ is extremely hard. Or is it?



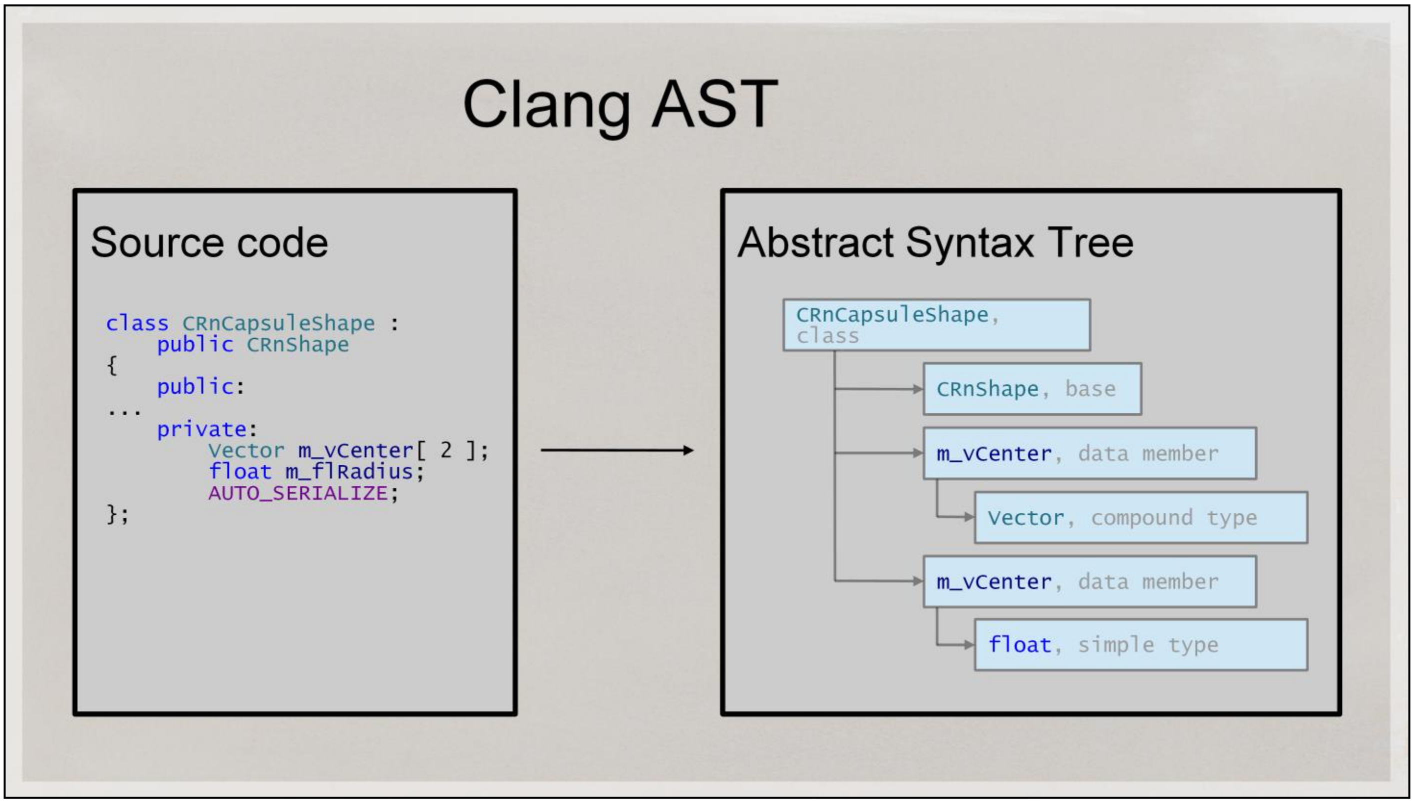
Clang to the Rescue!

C++ Classes

Clang

Serializer Code

- Clang is an open-source c++ compiler. In some ways it's a direct competitor of GCC, but with much more readable source code and liberal license.
- Clang makes C++ parsing a piece of cake. It's written in C++ itself, and it just works.
- We can use all our existing code as input, convert cpp and header files into easy-to-use AST and spit out the serializer routines.
- One bonus from using Clang is this improved C++ compliance of our code. We found some bugs, and prepared the code for Linux. It's really useful to compile your code with another compiler.



Clang AST

Source code

```
class CRnCapsuleShape :
    public CRnShape
{
    public:
...
    private:
        Vector m_vCenter[ 2 ];
        float m_flRadius;
        AUTO_SERIALIZE;
};
```

Abstract Syntax Tree

CRnCapsuleShape, class

CRnShape, base

m_vCenter, data member

Vector, compound type

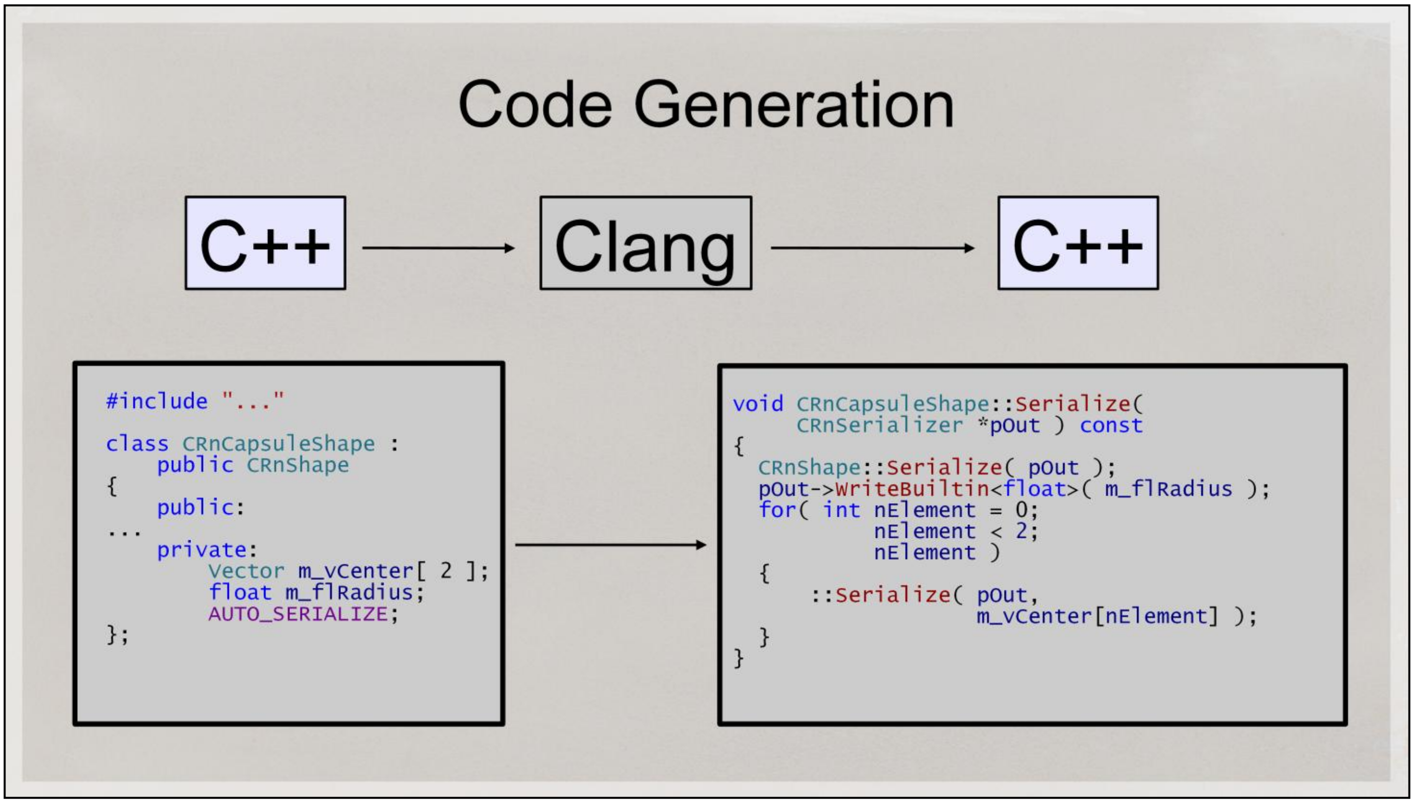
m_vCenter, data member

float, simple type

I mentioned the term AST. It means Abstract Syntax Tree.

It's a tree that describes everything there is to know about your source code.

This is how it looks, schematically. For every class, you'll have a node. For every member, its type or size, or annotation, or anything you can think of you'll probably get a node of some type.

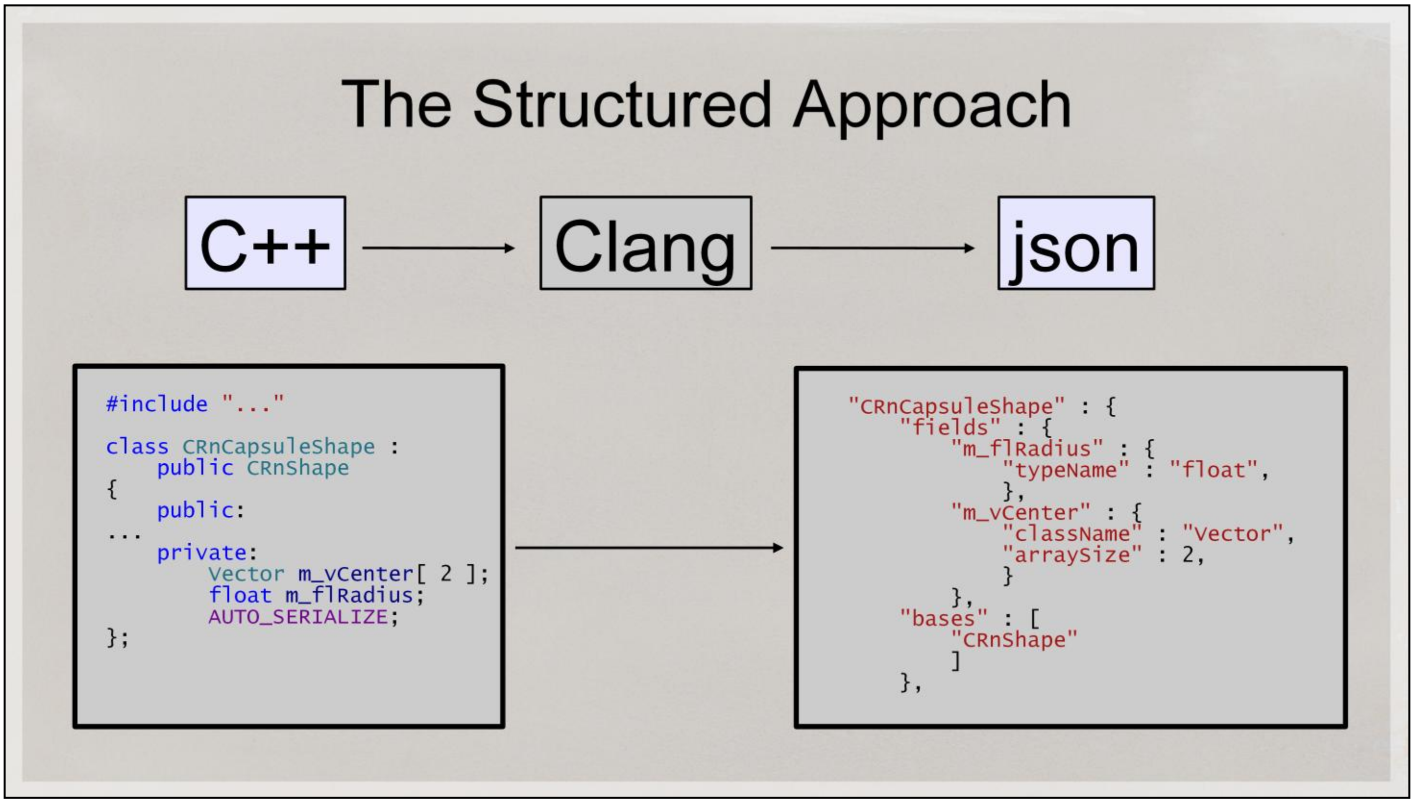


Code Generation

```
#include "..."

class CRnCapsuleShape :
    public CRnShape
{
    public:
...
    private:
        Vector m_vCenter[ 2 ];
        float m_flRadius;
        AUTO_SERIALIZE;
};
```

```
void CRnCapsuleShape::Serialize(
     CRnSerializer *pOut ) const
{
  CRnShape::Serialize( pOut );
  pOut->WriteBuiltin<float>( m_flRadius );
  for( int nElement = 0;
           nElement < 2;
           nElement )
  {
      ::Serialize( pOut,
                   m_vCenter[nElement] );
  }
}
```

- In effect, we are transforming C++ code into additional C++ code
- It's very tempting at first to put code generation as a bunch of printf() statement right into the your parser
- I decided to resist the urge, and I consider myself lucky I did. It would be horribly slow to iterate on the generated code with printfs.



The Structured Approach

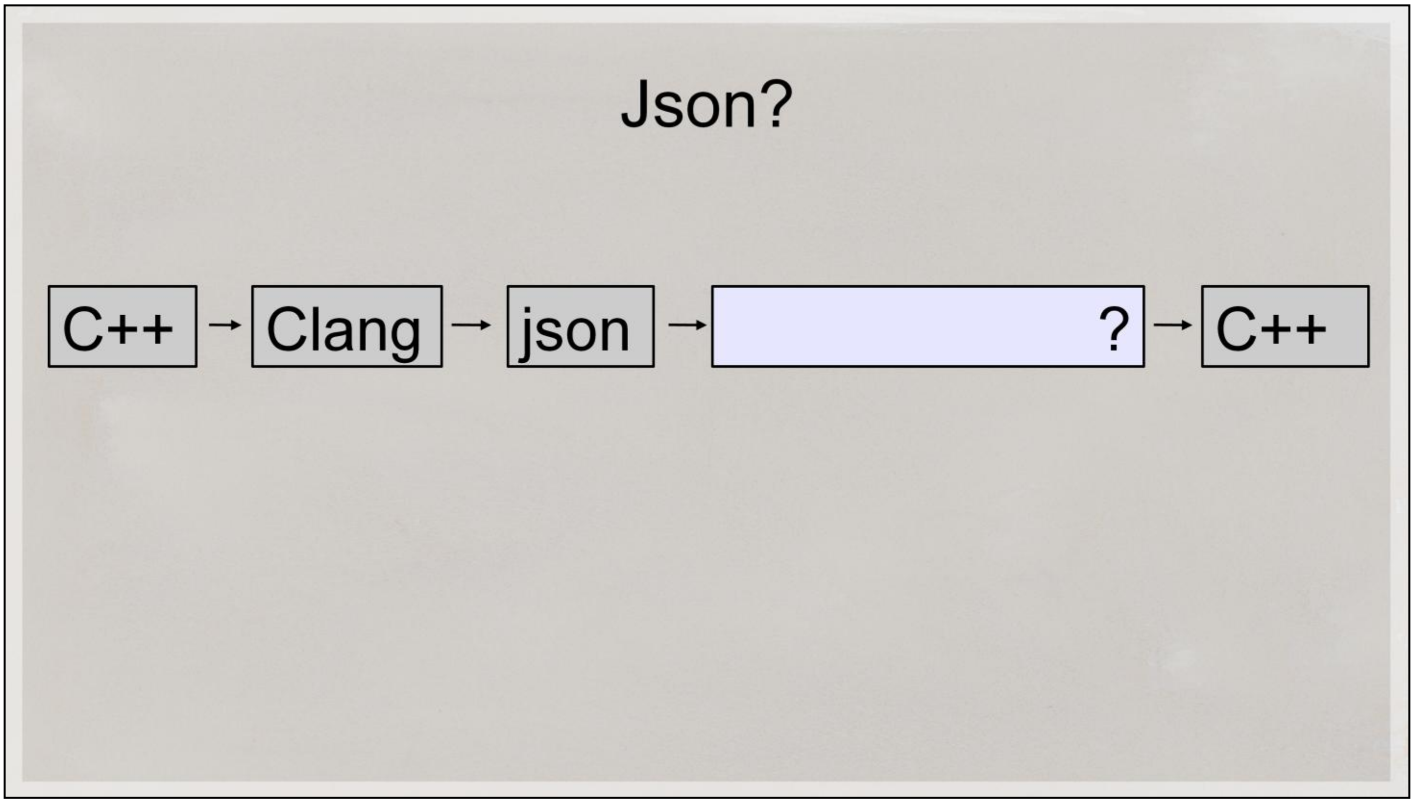
```
#include "..."

class CRnCapsuleShape :
    public CRnShape
{
    public:
...
    private:
        Vector m_vCenter[ 2 ];
        float m_flRadius;
        AUTO_SERIALIZE;
};
```

```
"CRnCapsuleShape" : {
    "fields" : {
        "m_flRadius" : {
            "typeName" : "float",
            },
        "m_vCenter" : {
            "className" : "Vector",
            "arraySize" : 2,
            }
        },
    "bases" : [
        "CRnShape"
        ]
    },
```

Parse, *then* Generate. Decouple these two stages, and unless your project is trivial, you'll be rewarded.

I'm distilling the huge AST of the physics engine into a very concise json file. Json is a human readable format that is widely used nowadays.

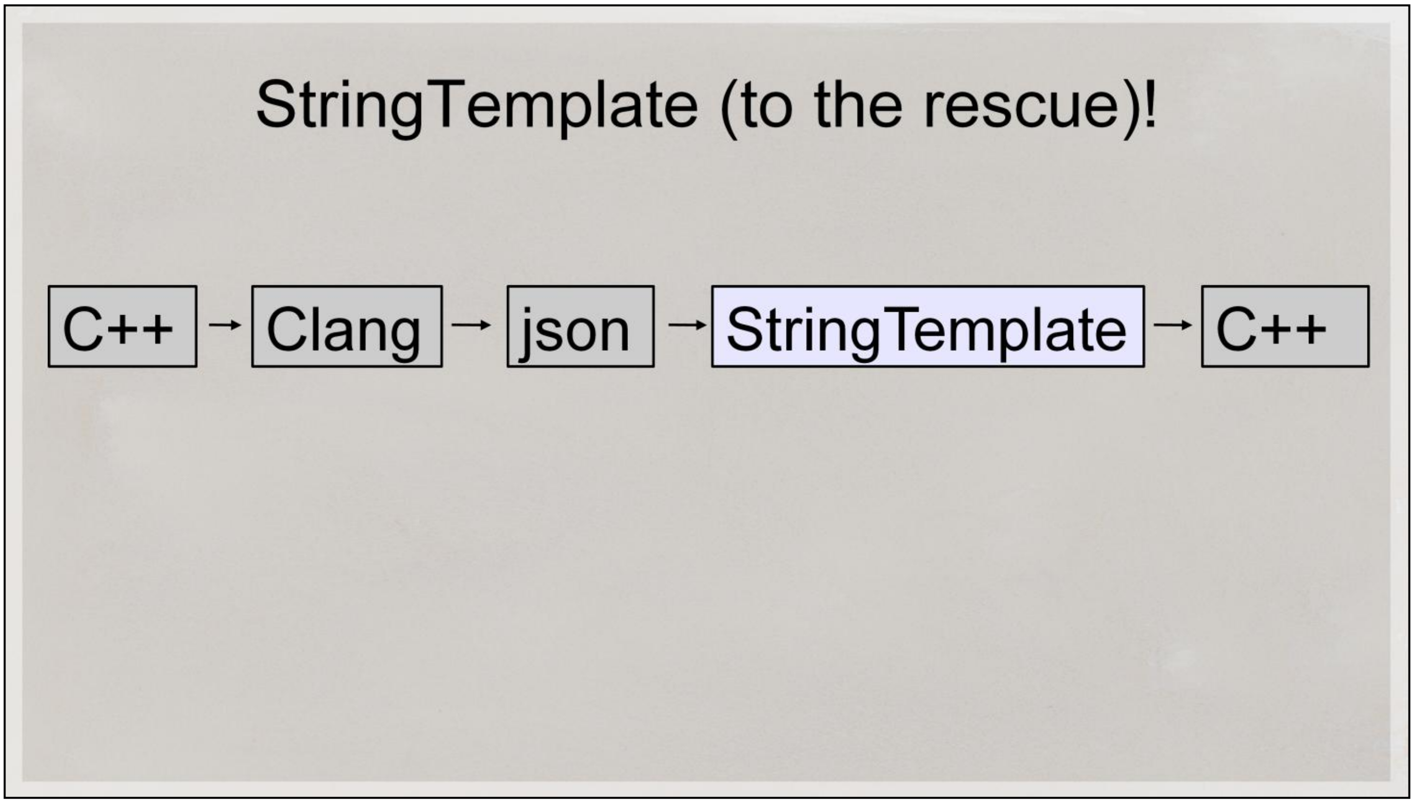
After adding features for a year, my Visitor class in the code that uses Clang that just prints out the json with relevant information is about 1000 lines long, and it would be incomprehensible if I generated code there.



Json?

C++ → Clang → json → ? → C++

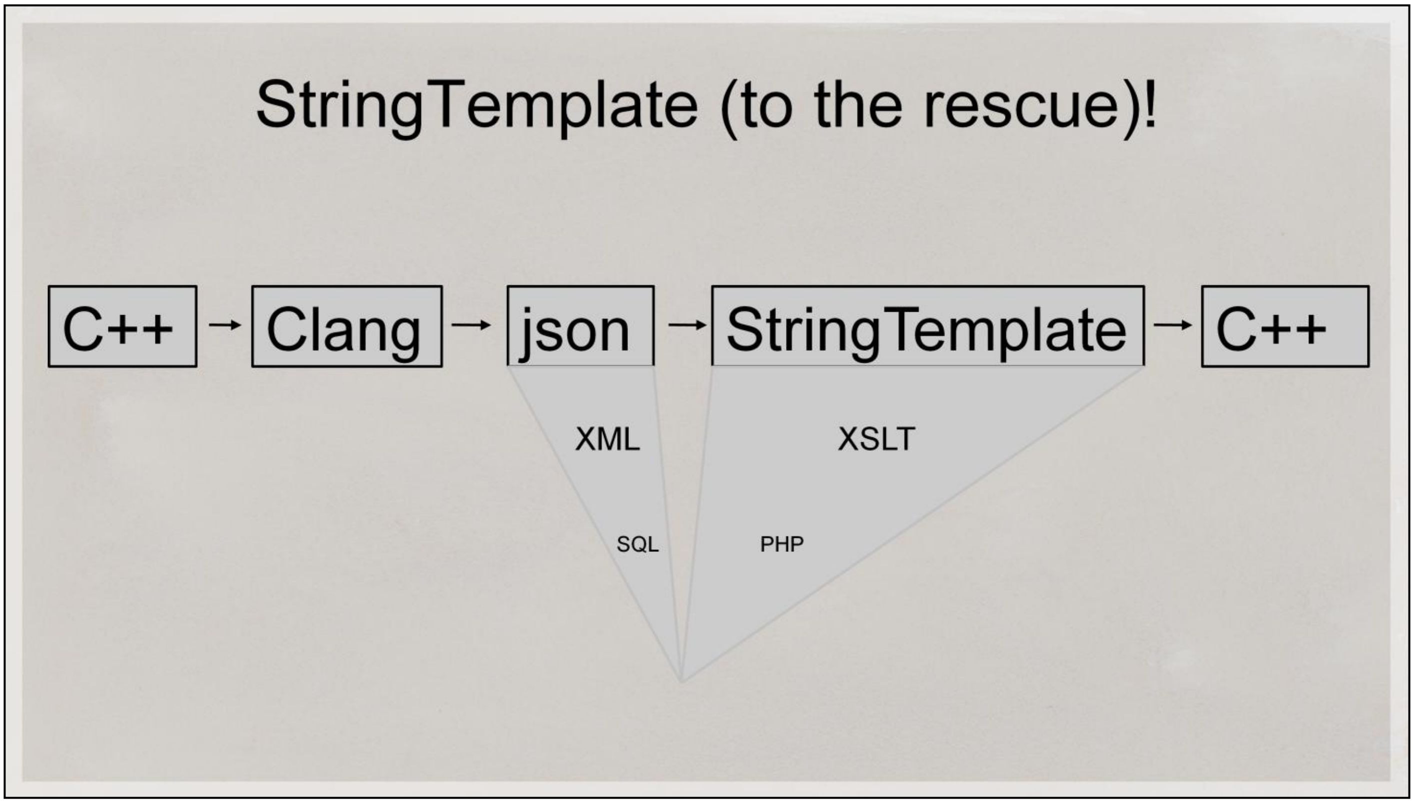
- So, I'm taking my C++ code, let Clang parse it and spit out Json.
- But I ultimately need my serializer and potentially other useful tools to be written in C++
- This begs the question: how do I generate C++ code from Json?



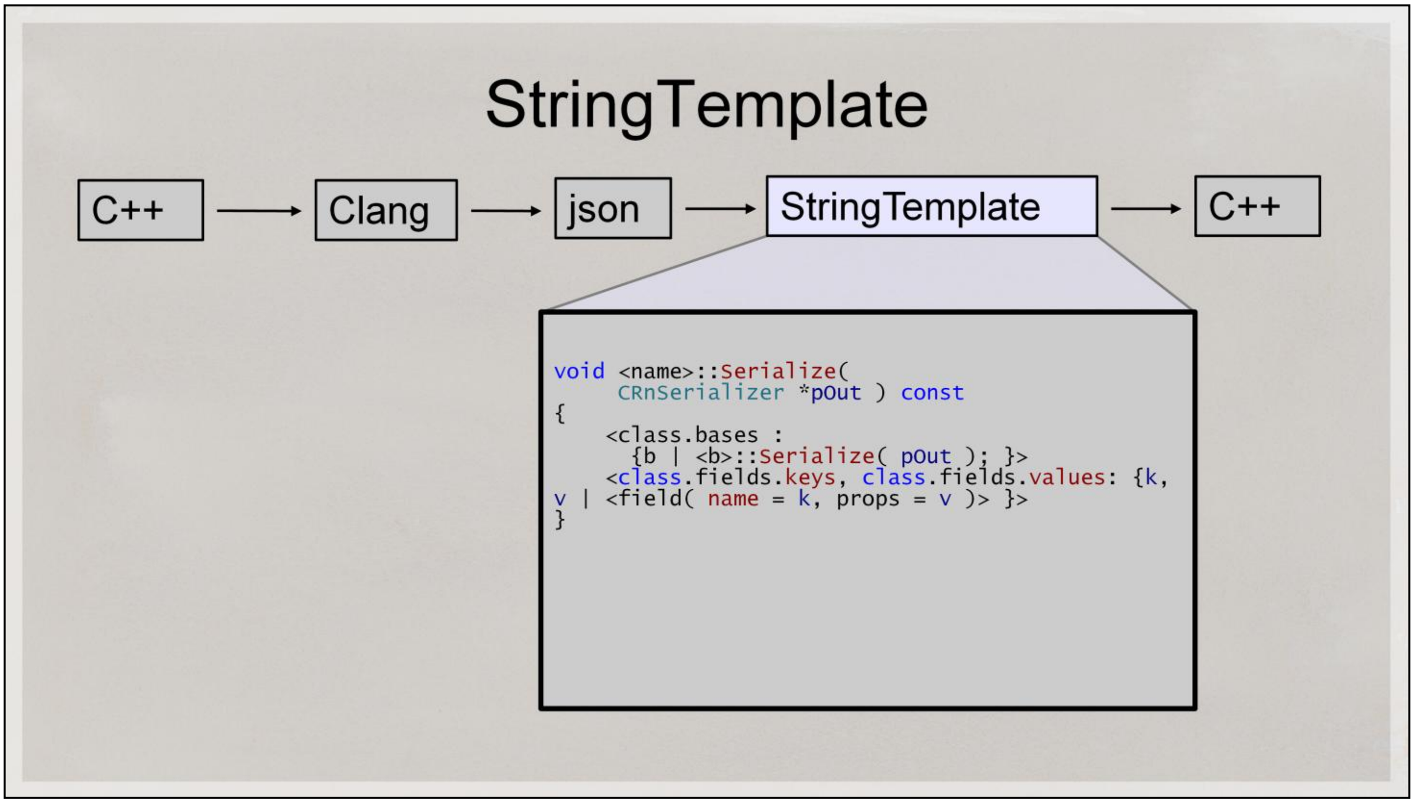
StringTemplate (to the rescue)!

C++ → Clang → json → StringTemplate → C++

- Fortunately, there is no need to reinvent the wheel, because it was invented many times over before us. There are a lot of text template libraries, and languages. They are widely used and freely available.
- I chose StringTemplate. It's a text library that's very easy to use, and it's very well suited to code generation. It's declarative style, which is appealing for this kind of task.


StringTemplate (to the rescue)!

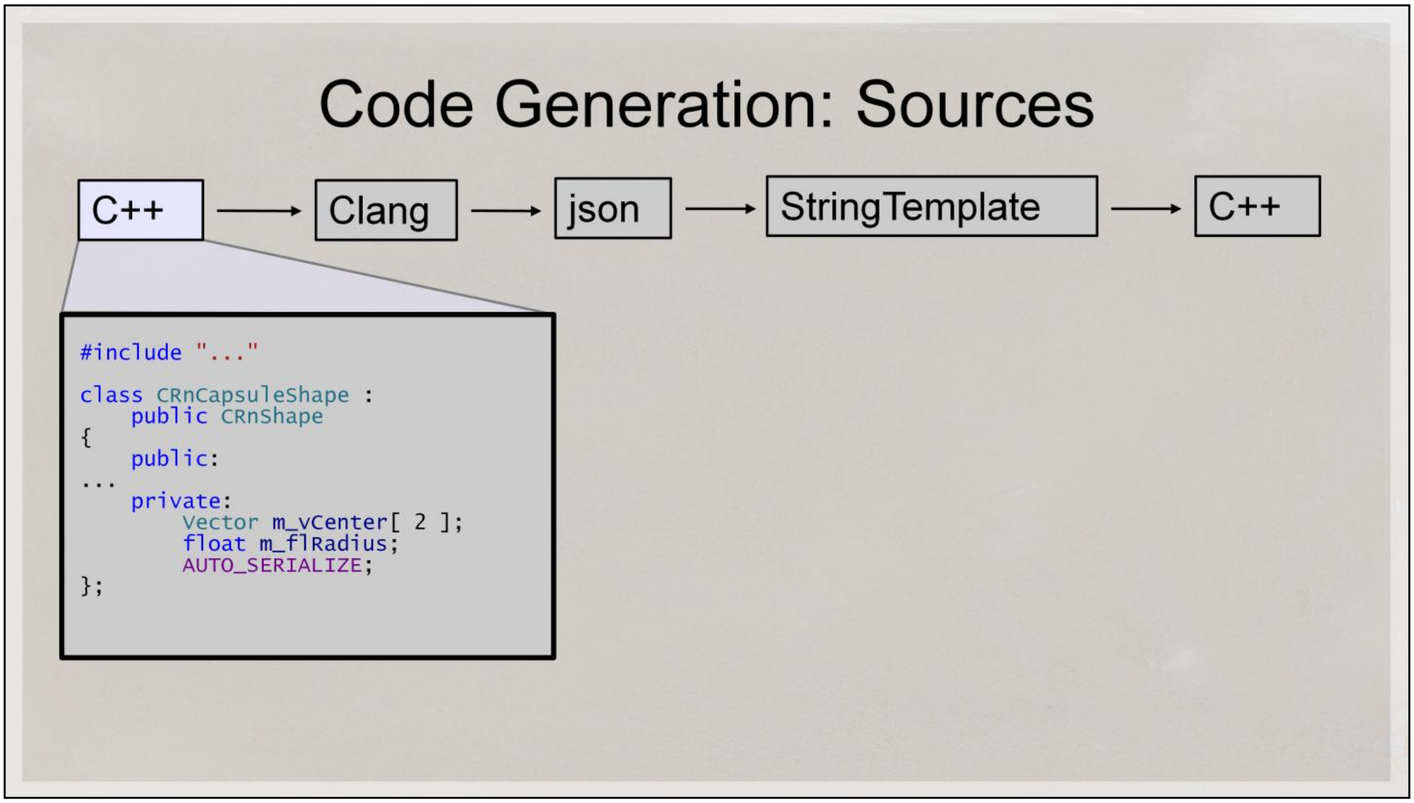
- You could use something else. In fact, if you're very familiar and comfortable with, say, PHP, you should absolutely use it.
- You could also use XML instead of Json. In fact, standard Clang tools can output full abstract syntax tree in XML directly. XSLT is a powerful transformation language for XML. StringTemplates and Json are much easier to learn and much more human-readable. That's why I personally chose them.
- There are many ways to configure this pipeline, I'm just describing what worked for me.



StringTemplate

```
void <name>::Serialize(
     CRnSerializer *pOut ) const
{
    <class.bases :
      {b | <b>::Serialize( pOut ); }>
    <class.fields.keys, class.fields.values: {k,
v | <field( name = k, props = v )> }>
}
```

- StringTemplate looks like this. The tool itself is written in java, but there's no need to know Java to use it. All my serialization, unserialization and memory statistics code is generated from a 400-line template. These are a few of those lines. They auto-generate the code to serialize most classes in our physics engine.
- You feed it json file, it spits out C++ file. Quite simple.

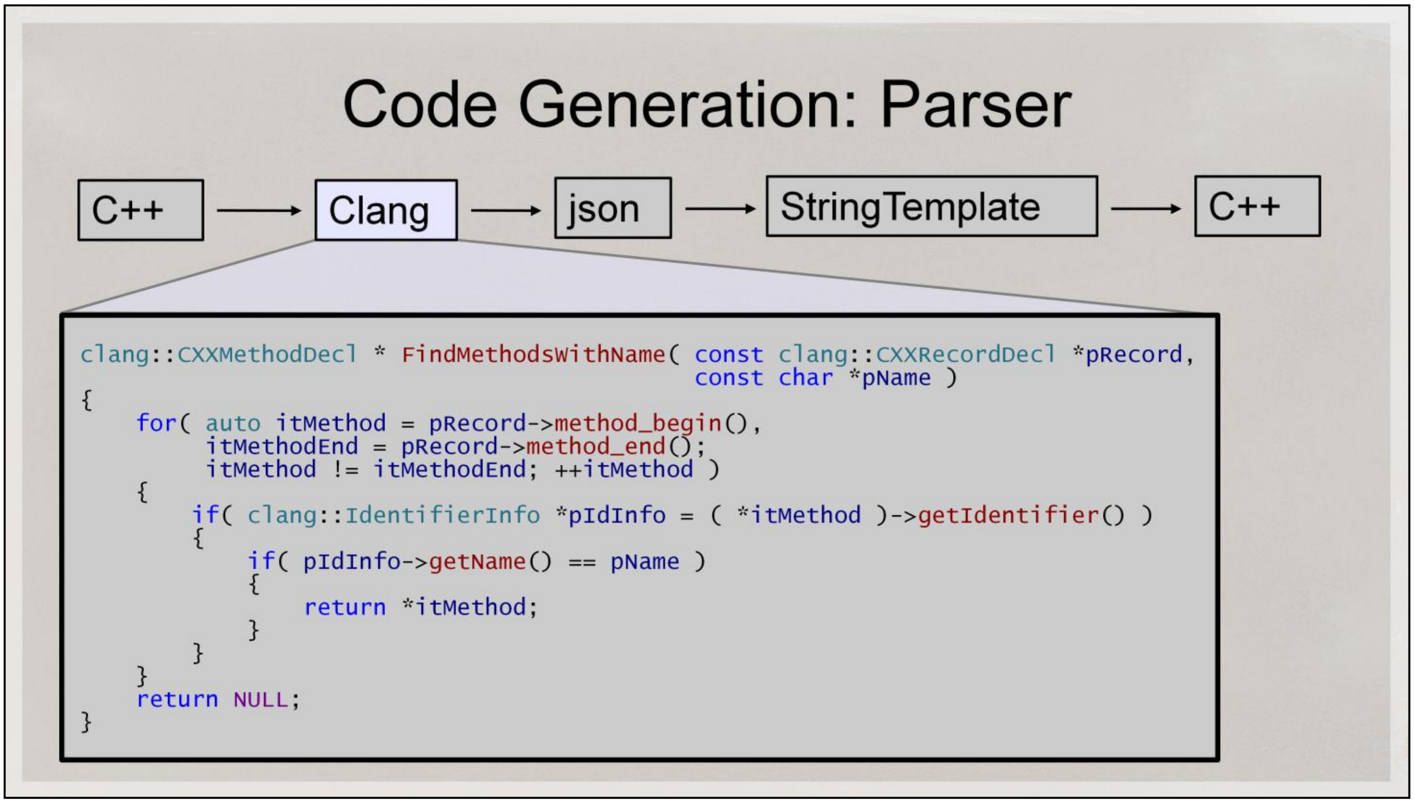


Code Generation: Sources

C++ → Clang → json → StringTemplate → C++

```
#include "..."

class CRnCapsuleShape :
    public CRnShape
{
    public:
...
    private:
        Vector m_vCenter[ 2 ];
        float m_flRadius;
        AUTO_SERIALIZE;
};
```

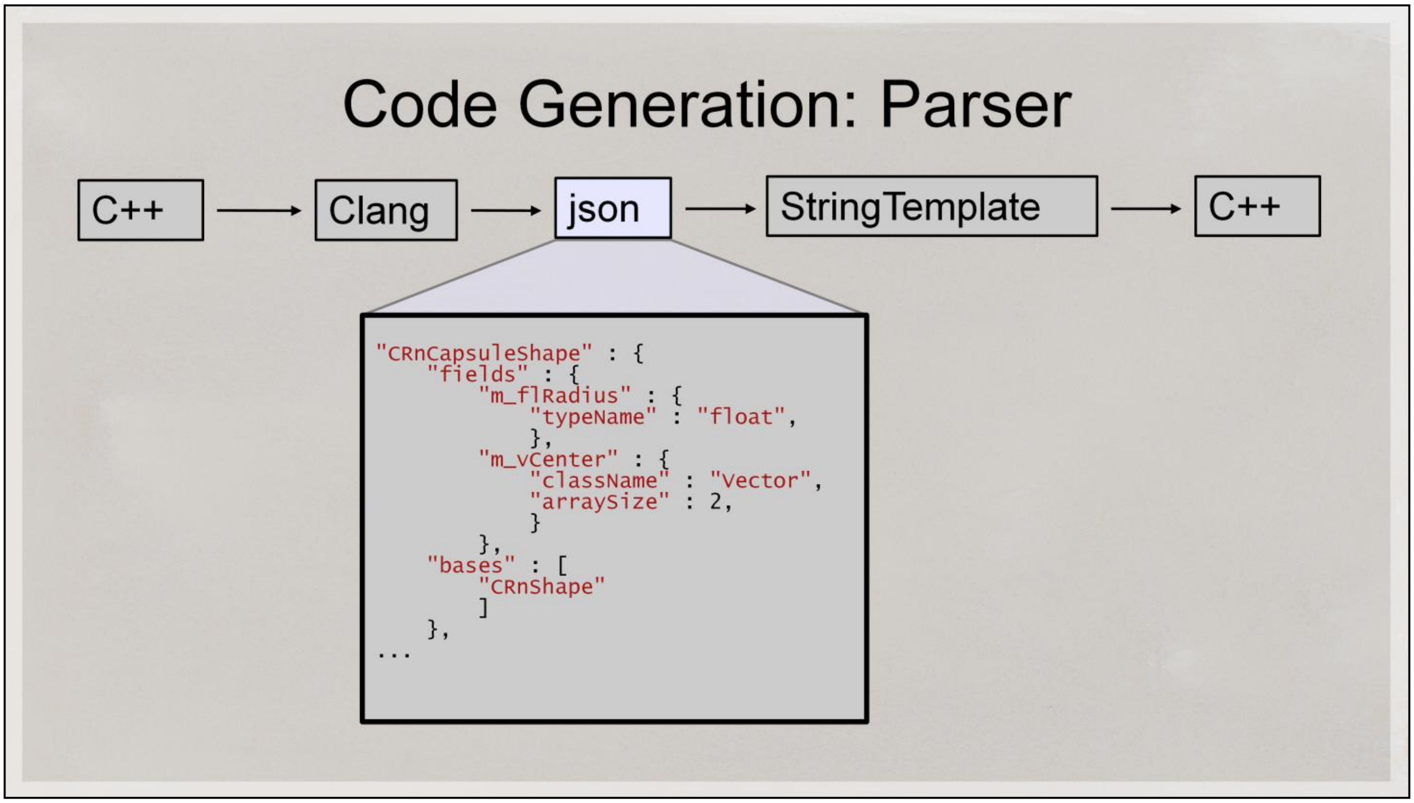
- So, to recap.
- We start with the source code of the engine – it's arbitrary C++, potentially annotated. For instance, you can annotate only the classes you want serialized. And exclude some fields from serialization. And add a member to call after serialization.



Code Generation: Parser

```
clang::CXXMethodDecl * FindMethodsWithName( const clang::CXXRecordDecl *pRecord,
                                            const char *pName )
{
    for( auto itMethod = pRecord->method_begin(),
         itMethodEnd = pRecord->method_end();
         itMethod != itMethodEnd; ++itMethod )
    {
        if( clang::IdentifierInfo *pIdInfo = ( *itMethod )->getIdentifier() )
        {
            if( pIdInfo->getName() == pName )
            {
                return *itMethod;
            }
        }
    }
    return NULL;
}
```

- The parser/Clang part is moderately complicated. I wrote the first working version within a couple of weeks, working on it part-time. But then I fiddled with it for well over a year. It handles all the special and edge cases we have in our engine. It's not a magic bullet, though. It will not parse anything and everything in the known universe, but it'll parse and describe any exotic data structure in our engine. I have to change it only very rarely. In fact, the last change I made was very minor. And it was over 4 months ago.
- This slide shows how to look for a method with a given name in class. It's pretty straightforward to extract information once you start writing the code.
- The best source of information for me was the online Clang docs, especially their class diagram. It's very logical and educational. Also, Clang is very readable, so a lot of stuff becomes clearer after you debug it.

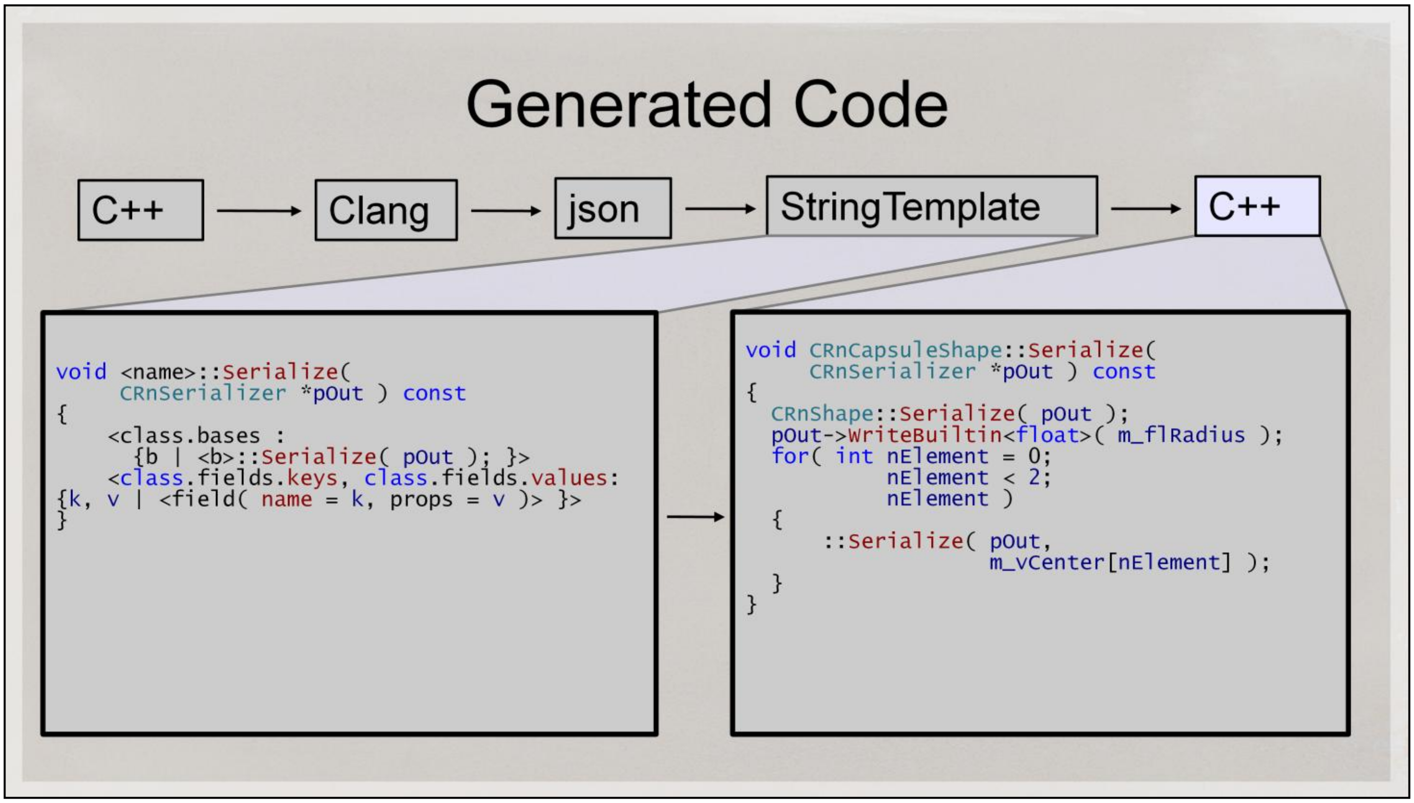


Code Generation: Parser

```
"CRnCapsuleShape" : {
    "fields" : {
        "m_flRadius" : {
            "typeName" : "float",
            },
        "m_vCenter" : {
            "className" : "Vector",
            "arraySize" : 2,
            }
        },
    "bases" : [
        "CRnShape"
        ]
    },
...
```

- The generated json file looks like this. I already explained the meaning of this snippet before.

- ~~It contains all the data necessary to generate the serializer and whatever other toold you might wish to generate, and nothing else. It's very much human-readable, which helps a lot when developing it~~
- ~~On the slide, it shows the class name, the list of its fields with field names and types and if the field is an array, then the array count. There's also the base class name. It's what you would expect.~~



Generated Code

C++ → Clang → json → StringTemplate → C++

```
void <name>::Serialize(
     CRnSerializer *pOut ) const
{
    <class.bases :
      {b | <b>::Serialize( pOut ); }>
    <class.fields.keys, class.fields.values:
{k, v | <field( name = k, props = v )> }>
}
```

```
void CRnCapsuleShape::Serialize(
     CRnSerializer *pOut ) const
{
  CRnShape::Serialize( pOut );
  pOut->WriteBuiltin<float>( m_flRadius );
  for( int nElement = 0;
           nElement < 2;
           nElement )
  {
      ::Serialize( pOut,
                   m_vCenter[nElement] );
  }
}
```

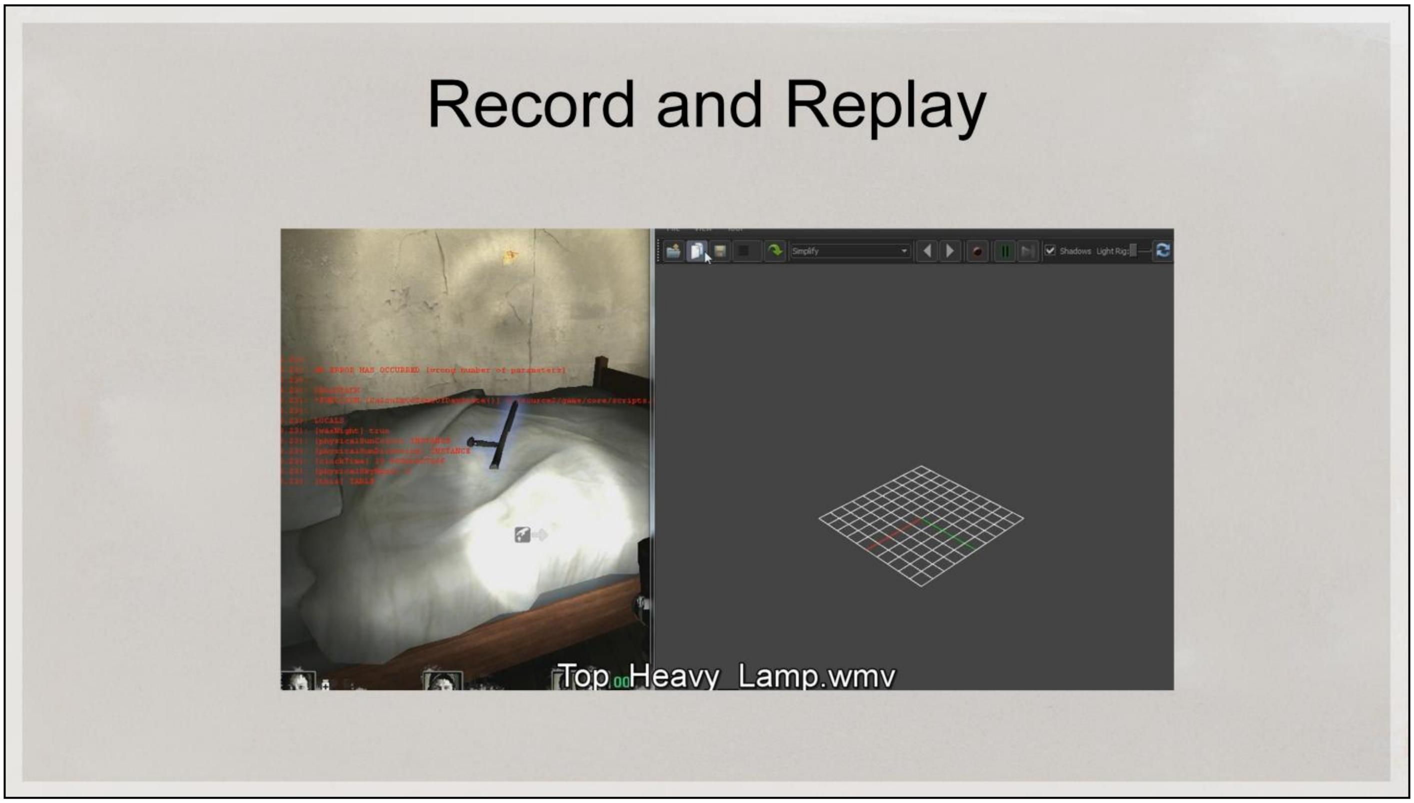
- To remind, this is what the string template looks like, and this here is the resulting generated C++ code
- The template goes through all data members of the class and generates code for each of them.
- The template is smart enough to know where I have arrays, templates, pointers to follow and so on.
- All the non-plain data elements are recursively serialized by other overloaded serialization functions. Some of them methods, some of them free functions. I especially prefer free functions for simpler types, containers and third party classes where I don't want to touch the header where those classes are declared.

Once you complete these 5 easy steps, you'll have yourself one wholesome serializer. It will be self-maintaining. You will forever be able to visualize your physics in all detail.

Sometimes we don't even know something is wrong. Some bugs ship and go unnoticed for years. This will never happen to you again.

Well..., I can't make guarantees... But I really hope I convinced you that visualizing stuff is great. Having serializer auto-magically generate itself makes everything even better.

But we have quite a bit more to discuss before we wrap up.



Record and Replay

```
[illegible] AN ERROR HAS OCCURRED [wrong number of parameters]
[illegible]
LOCALS
[wasRight] true
[illegible]
[table] TABLE
```

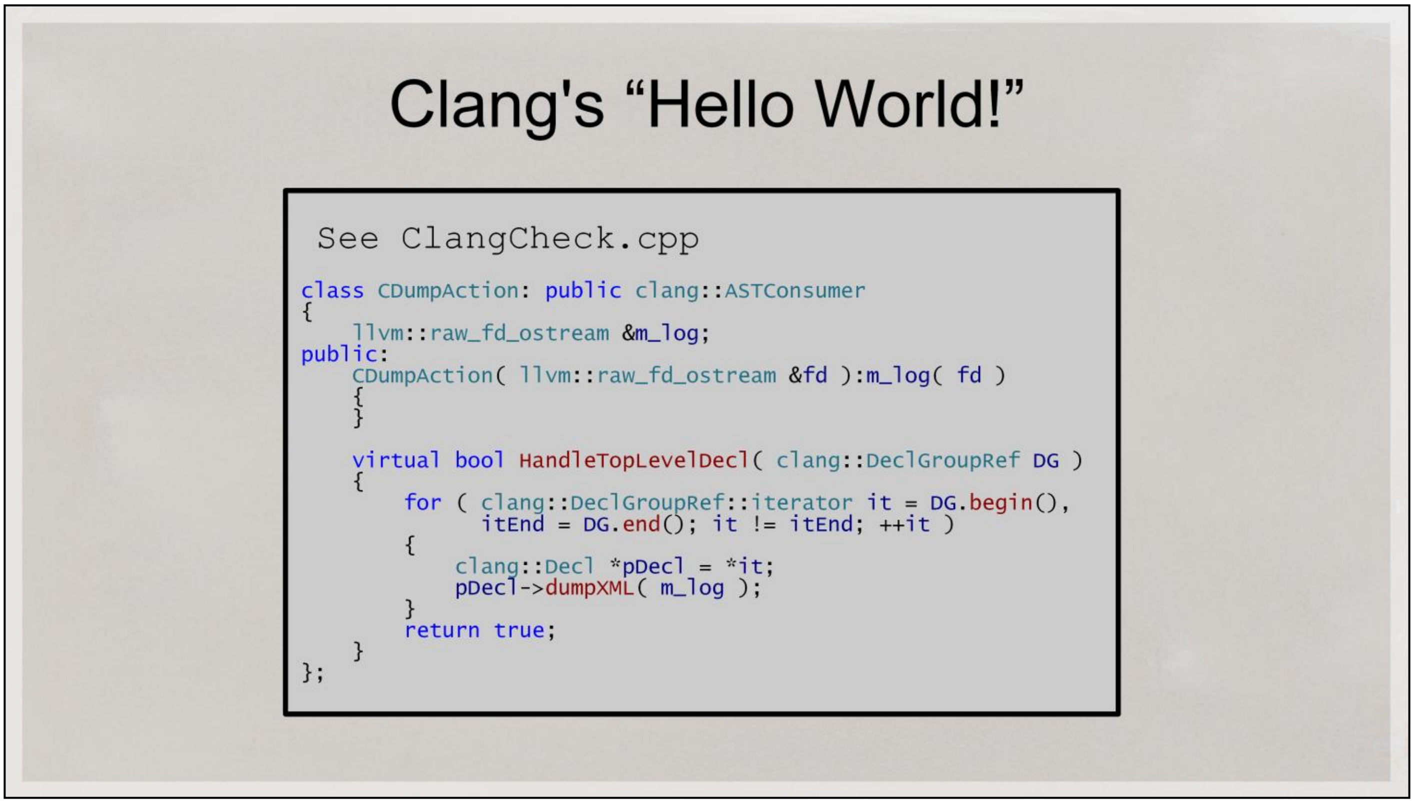
=Play= Top_Heavy_Lamp.wmv

Here's an example of one workflow when you have full serialization.

I noticed something weird with a lamp here. It was always falling down. Lamps are supposed to stand straight, and this one was always on its side.

So I snooped and looked at the lamp.

(wait) When I figured out why it's not standing (because its center of mass is too high), I just sent out an email to an artists – and I attached the file he can play with to prove my words. It's quite intuitive.



Clang's “Hello World!”

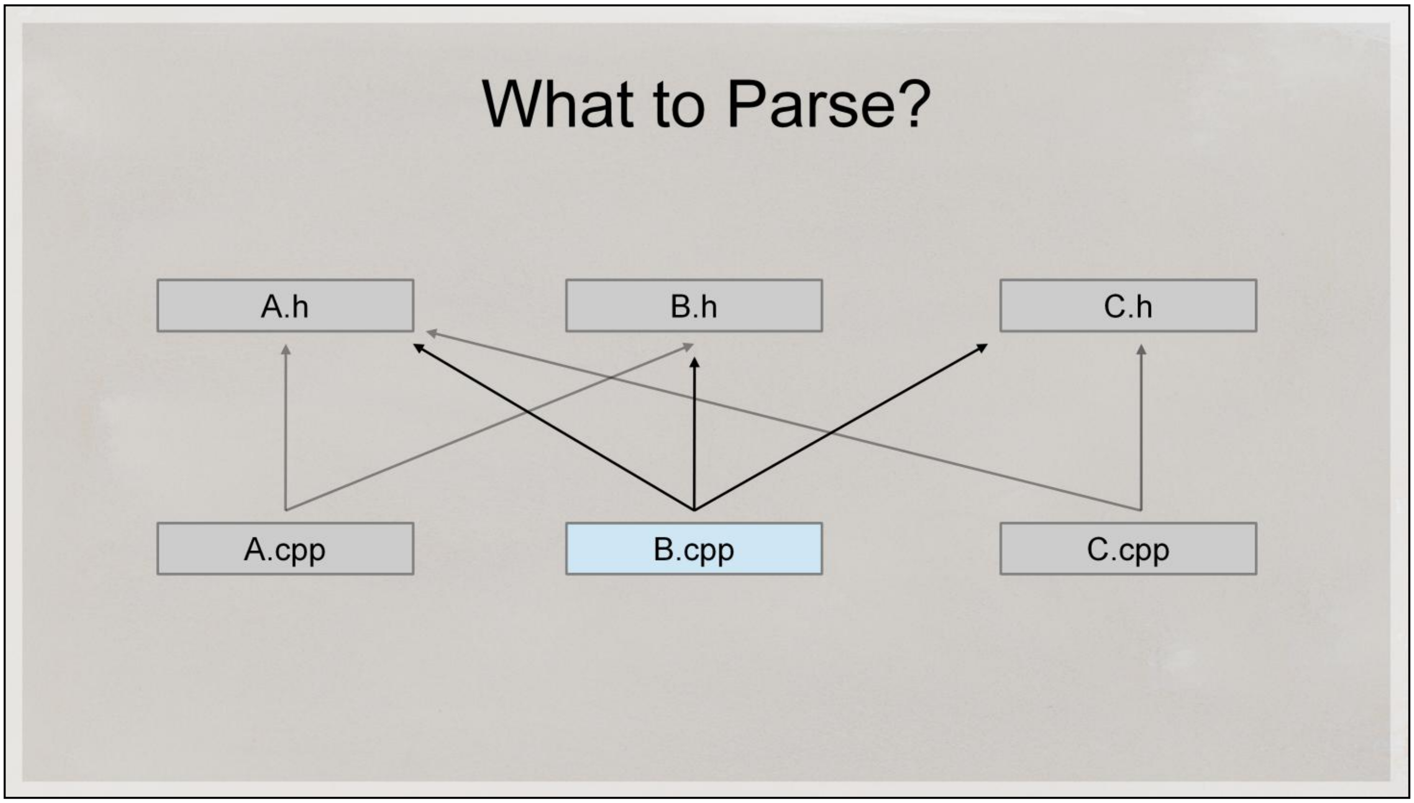
See ClangCheck.cpp

```
class CDumpAction: public clang::ASTConsumer
{
    llvm::raw_fd_ostream &m_log;
public:
    CDumpAction( llvm::raw_fd_ostream &fd ):m_log( fd )
    {
    }

    virtual bool HandleTopLevelDecl( clang::DeclGroupRef DG )
    {
        for ( clang::DeclGroupRef::iterator it = DG.begin(),
              itEnd = DG.end(); it != itEnd; ++it )
        {
            clang::Decl *pDecl = *it;
            pDecl->dumpXML( m_log );
        }
        return true;
    }
};
```

Here is the Clang's “Hello World” program relevant to this use case is called ClangCheck and it is a great starting point to write your own tool. It's part of the Clang distro, and I urge you to have a look at it.

It uses visitor pattern. For every top-level declaration in your source file, such as a class or function definition, or a typedef, or a global, you get a callback. In that callback, you can drill down the AST.

This self-contained example is dumping those declarations as XML to a log. If XML is your thing and performance is not the most critical issue, this here just may be your parser.



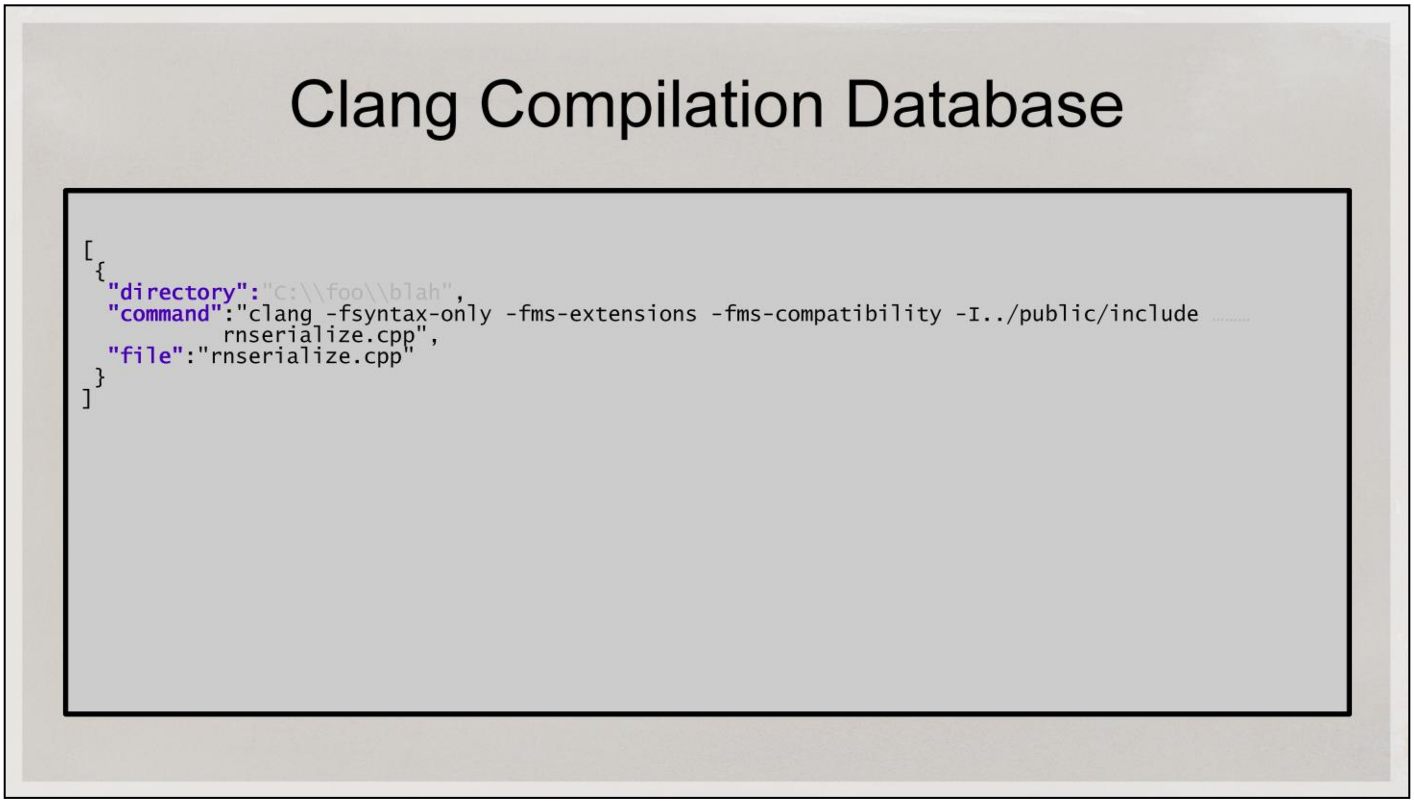
What to Parse?

Typically, all our class declarations are contained in headers.

It's enough to parse one .cpp file that #includes all the relevant headers for serialization.

There's no need to parse the whole engine, although there may be other useful tools that would need it.

To parse just one file, you'll need to create a compilation database file with cpp file name and compile options.



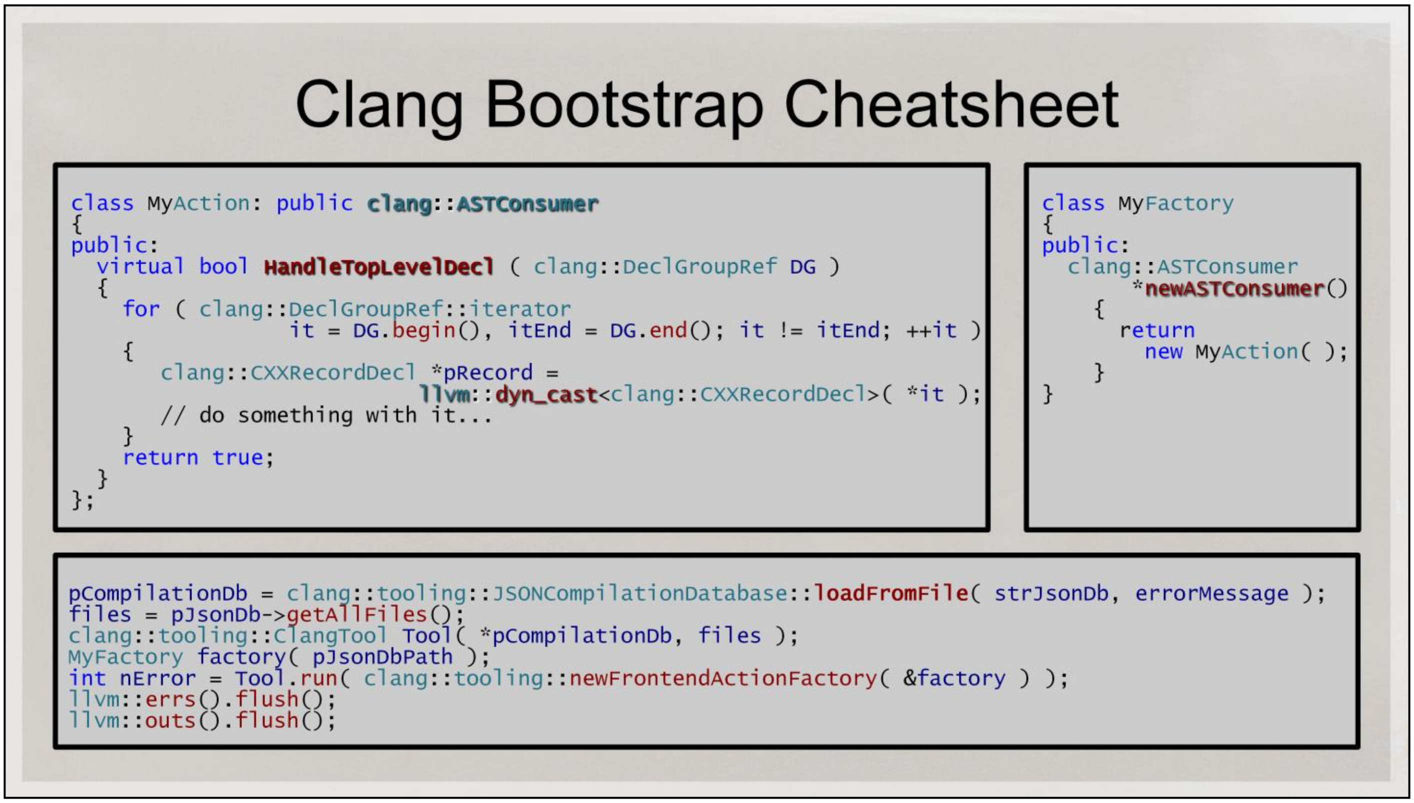
Clang Compilation Database

```
[
 {
  "directory":"C:\\foo\\blah",
  "command":"clang -fsyntax-only -fms-extensions -fms-compatibility -I../public/include ........
           rnserialize.cpp",
  "file":"rnserialize.cpp"
 }
]
```

Here's an example of compilation database. The most interesting options are highlighted here.

Compilation database is like a makefile, or vcproj. If you don't want to do anything non-standard, you can simply generate a .json file with all the information and give it to clang.

We use a tool to generate VCPROJ files, and we changed that tool to also spit out compilation databases.



Clang Bootstrap Cheatsheet

```
class MyAction: public clang::ASTConsumer
{
public:
  virtual bool HandleTopLevelDecl ( clang::DeclGroupRef DG )
  {
    for ( clang::DeclGroupRef::iterator
                it = DG.begin(), itEnd = DG.end(); it != itEnd; ++it )
    {
       clang::CXXRecordDecl *pRecord =
                          llvm::dyn_cast<clang::CXXRecordDecl>( *it );
       // do something with it...
    }
    return true;
  }
};
```

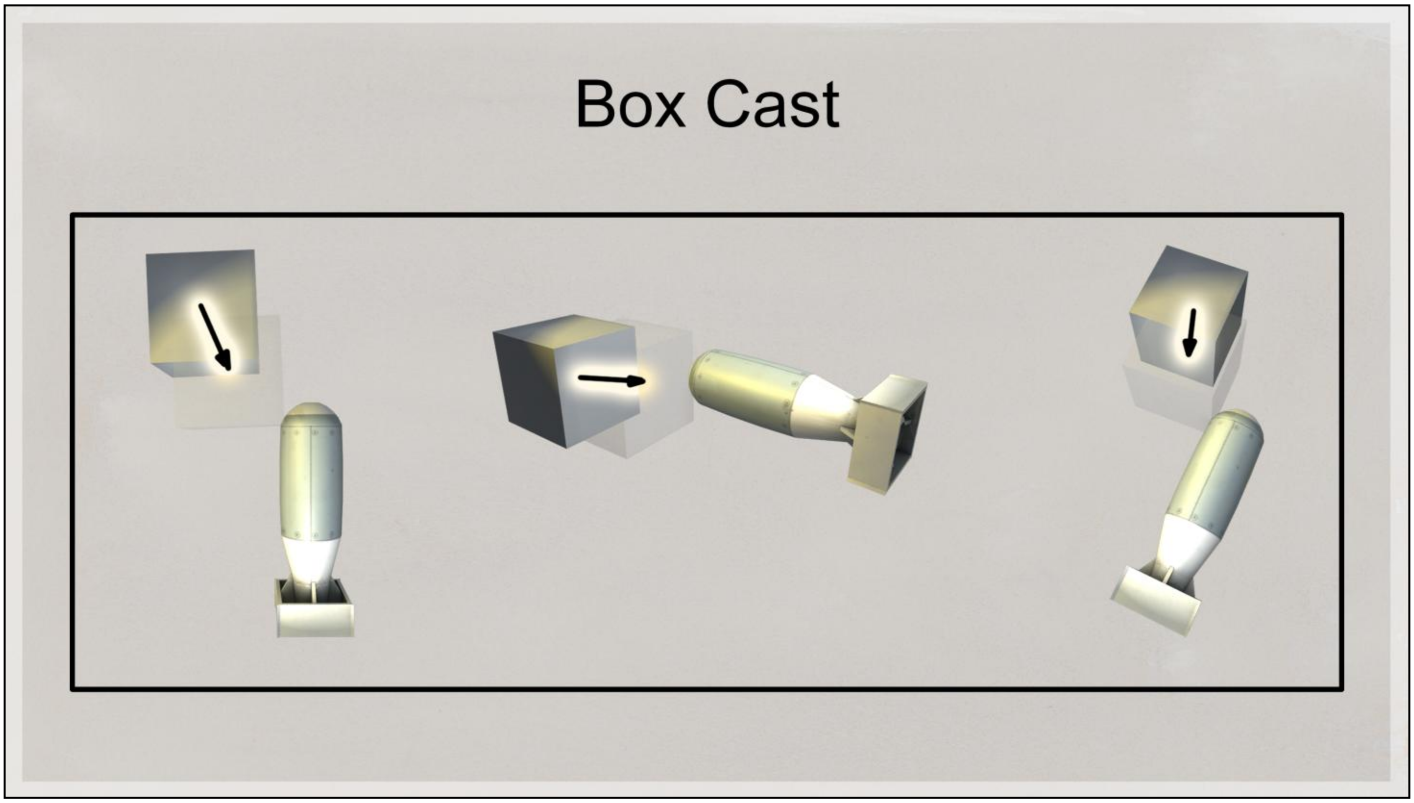
```
class MyFactory
{
public:
  clang::ASTConsumer
       *newASTConsumer()
    {
      return
        new MyAction( );
    }
}
```

```
pCompilationDb = clang::tooling::JSONCompilationDatabase::loadFromFile( strJsonDb, errorMessage );
files = pJsonDb->getAllFiles();
clang::tooling::ClangTool Tool( *pCompilationDb, files );
MyFactory factory( pJsonDbPath );
int nError = Tool.run( clang::tooling::newFrontendActionFactory( &factory ) );
llvm::errs().flush();
llvm::outs().flush();
```

And here is a little cheatsheet. In your parser tool, that you carefully link with all the clang libraries, what do you do first?

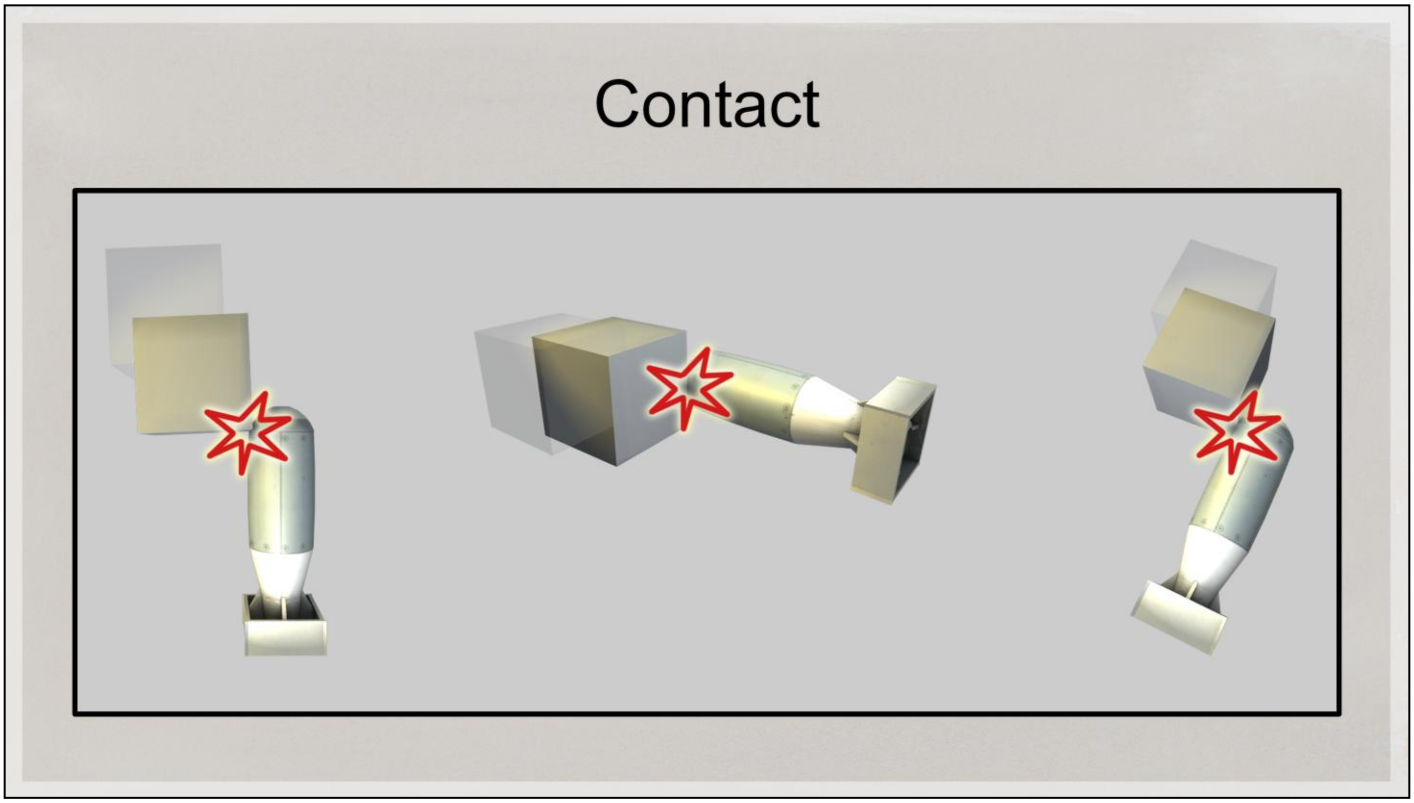
Start with implementing ASTConsumer, make a factory class for it, and then in your main() function, load compilation database and then run an instance ClangTool. It's all here in this slide.

One thing to remember: you may have to flush std out and std error.



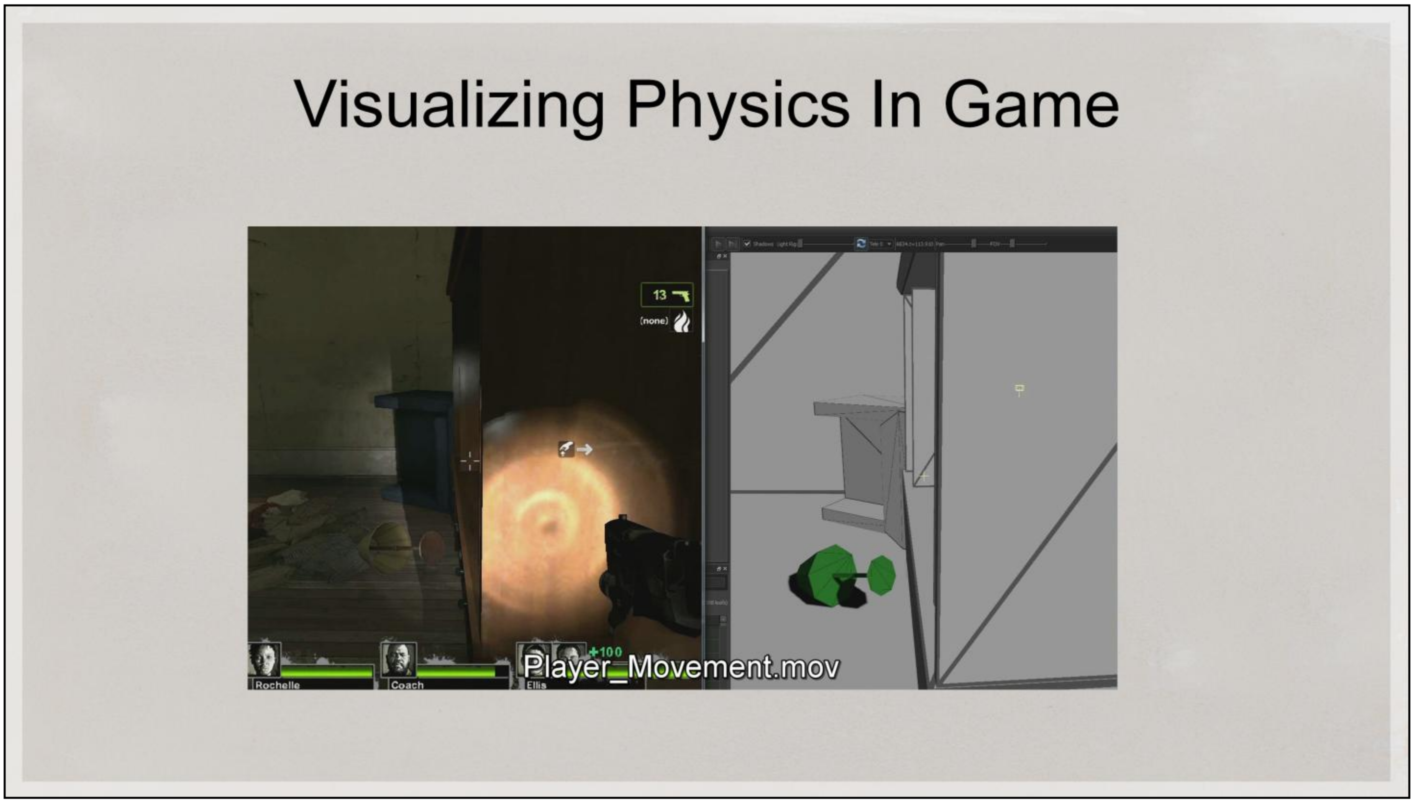
Box Cast

Before going further, I'd like to clarify what I mean by box cast.
We often use it in Source engine. It's like raycast, but with a box.



Contact

Just take a box, move it linearly until you hit something.
At that point, you sometimes create a contact point for the solver.
We use it for player movement a lot, so debugging it is a common thing to do.



Visualizing Physics In Game

Player_Movement.mov

=Run= Player_Movement.mov

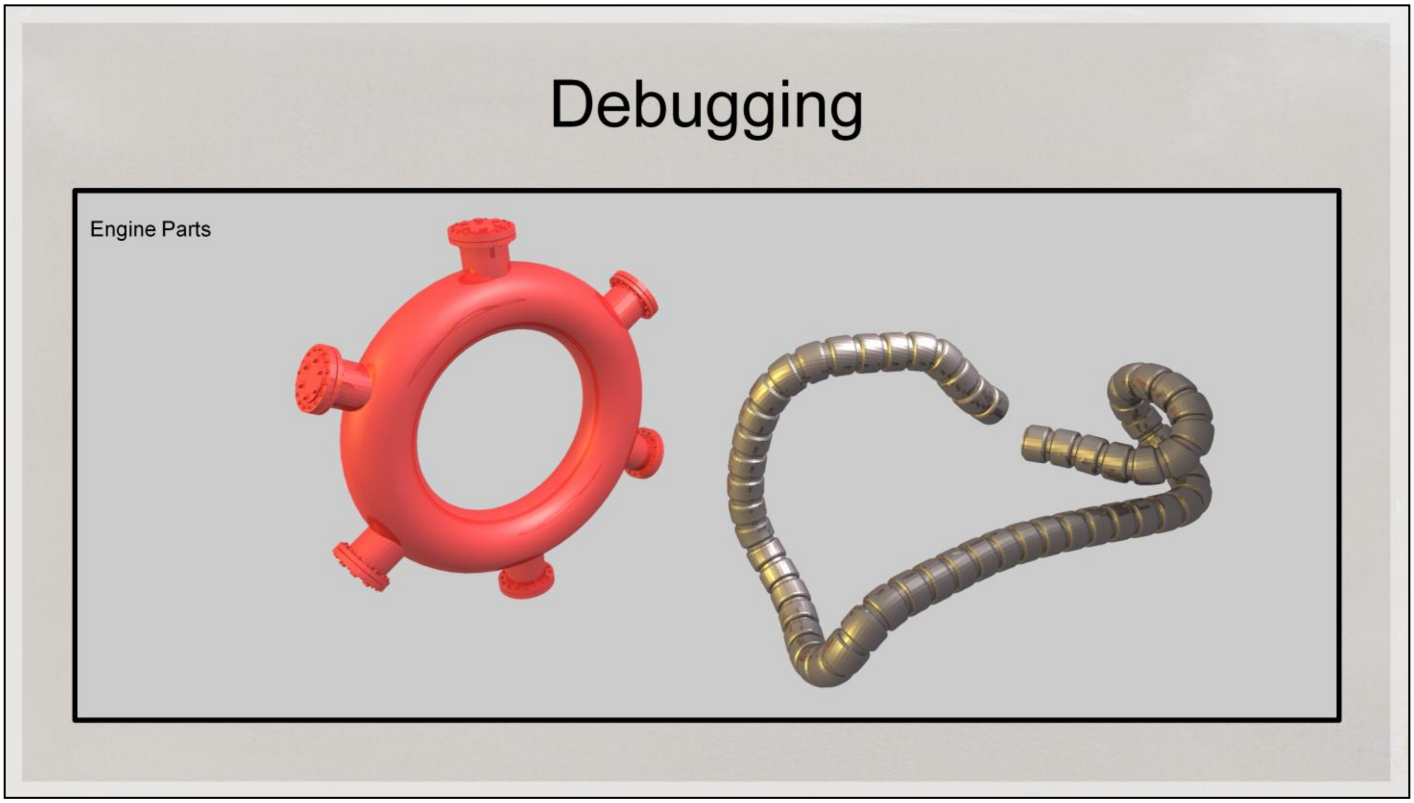
Let's see what we've got for our efforts so far.

Here's a game running alongside visualization tool.
I'm walking around a room, and I'm looking at the raycasts and box casts our player movement is making
I can look at it from the 3rd person view, I can only render the relevant data.
I'm noticing something strange, there's a flurry of box casts sometimes. It might probably result in a framerate hitch.
So I'm trying to reproduce this.
I find a place where it's happening.
And I'm flying around to see what causes it.
So far, so good. By the way, I can play with that physics at any point, but it won't affect the game, so it's not useful in this case.

When That Is Not Enough

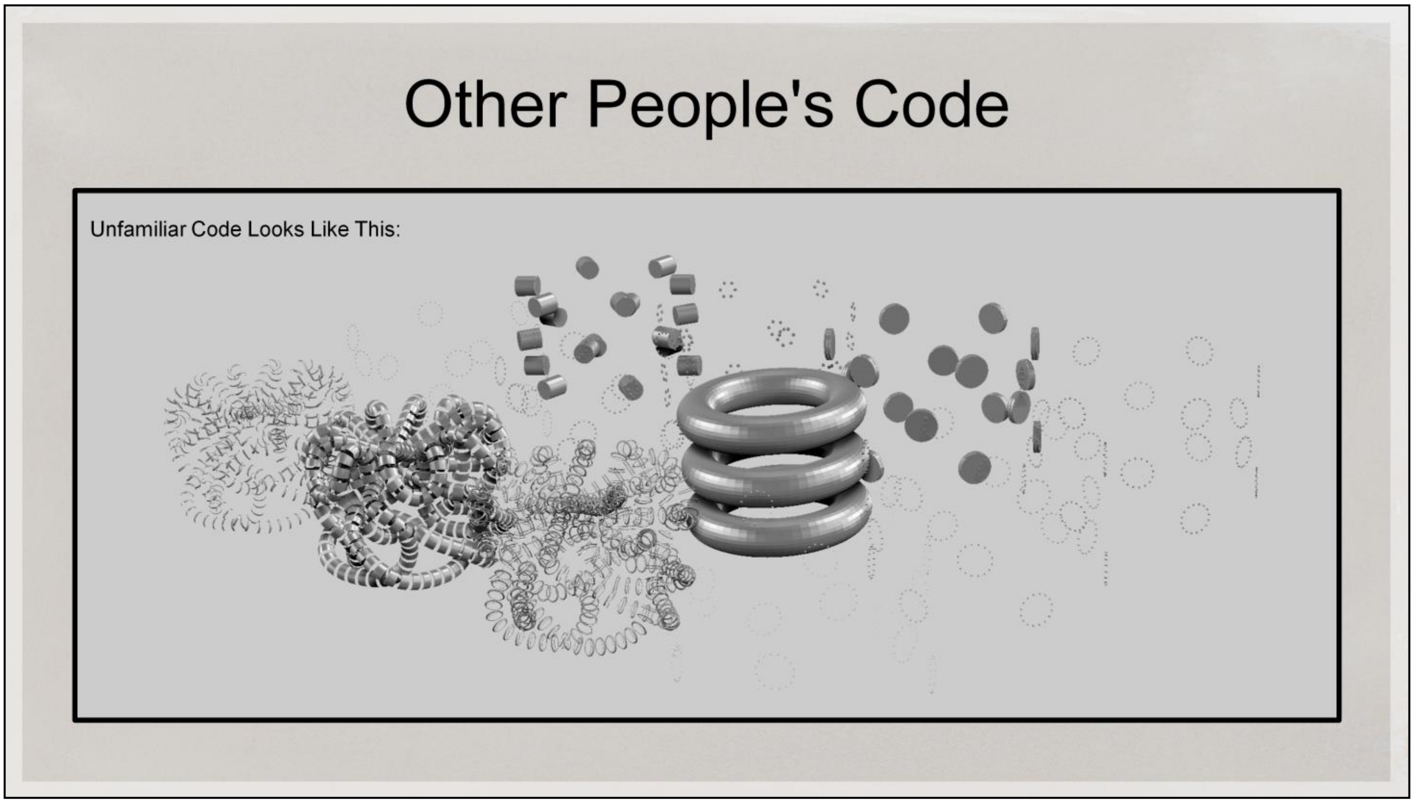
Well, what if just looking at the problem does not present an obvious fix?

What if we have to actually debug code?



Debugging

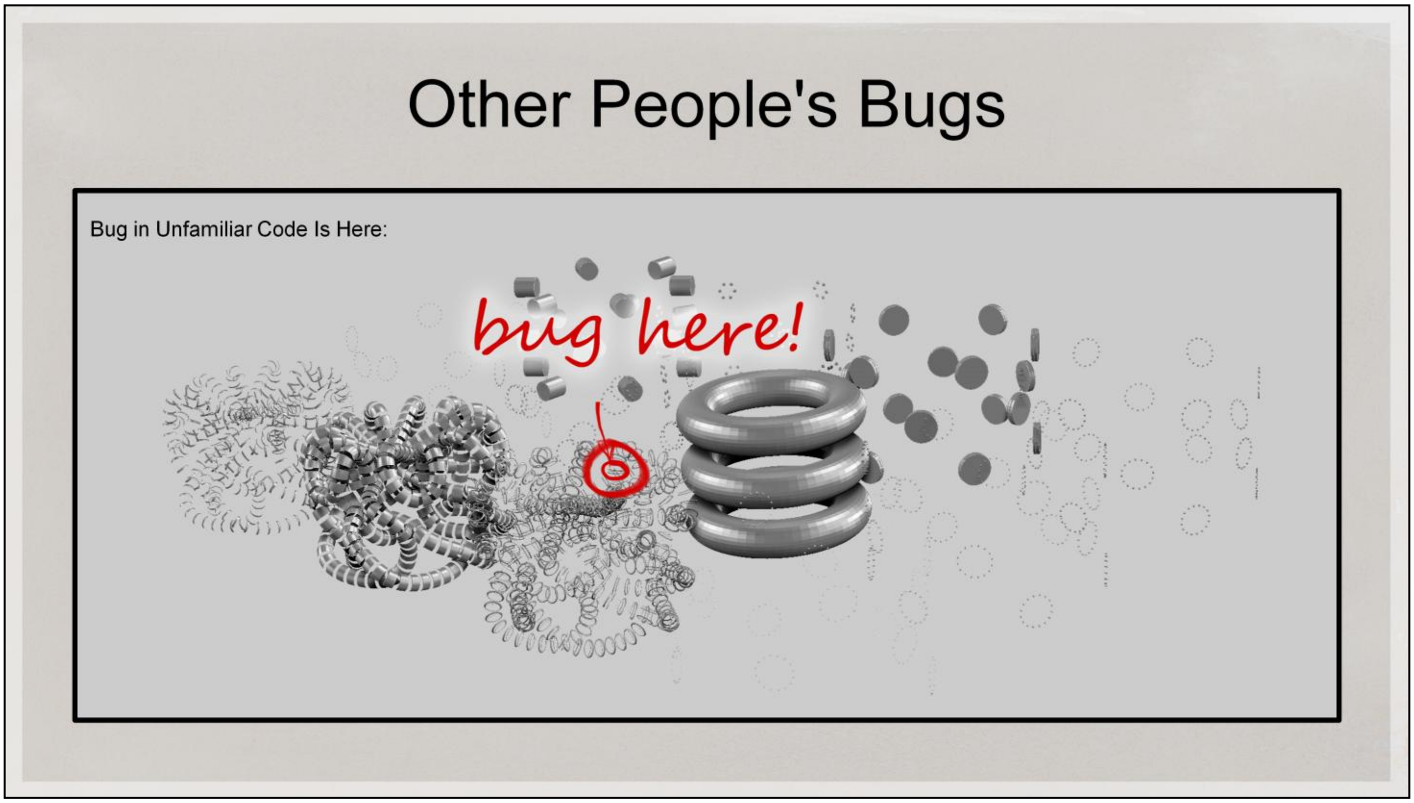
At least when you debug your own code, it's easy to see the parts. The components.
What they are supposed to do.
It's easy to check and see if they are actually doing it.



Other People's Code

Unfamiliar Code Looks Like This:

Working in a team, it's different... It's hard to see the whole picture. It's hard to check if the parts are working properly.



Other People's Bugs

Bug in Unfamiliar Code Is Here:

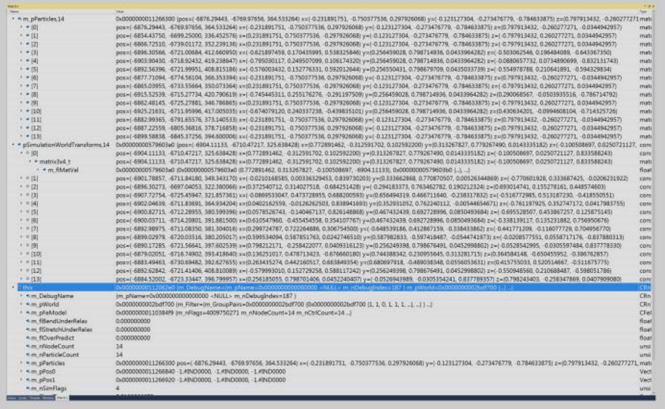
Even when I find a bug in foreign code, there are usually many ways to fix it and I'm never quite sure which one is the best.

Or maybe I misunderstood the code and my fix will cause another bug.

Tools

Tools of the trade

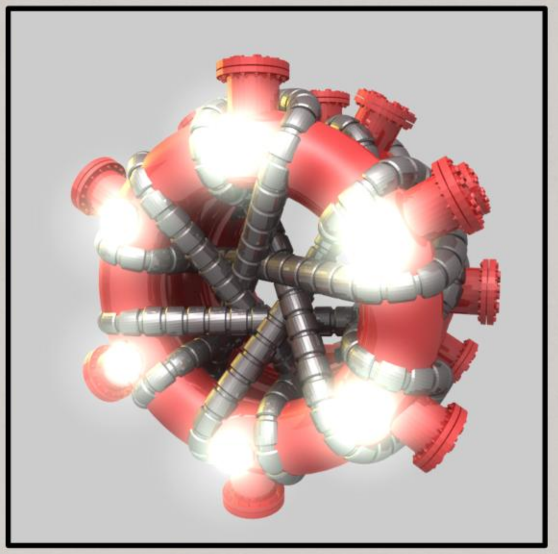
Watch Window

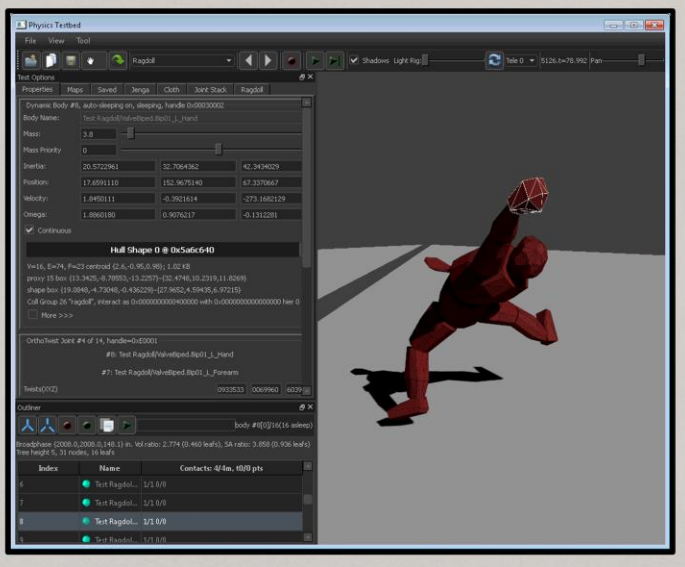


```
Name                           Value                                                                                                                                                                                      Type
▾ m_pParticles,14              0x0000000011266300 {pos={-6876.29443, -6769.97656, 364.533264} x={-0.231891751, -0.750377536, 0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={0.797913432, -0.260277271  mat
  ▸ [0]                        pos={-6876.29443, -6769.97656, 364.533264} x={-0.231891751, -0.750377536, 0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={0.797913432, -0.260277271, -0.0344942957}  mat
  ▸ [1]                        pos={-6854.43750, -6699.25000, 336.452576} x={0.231891751, 0.750377536, -0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={-0.797913432, 0.260277271, 0.0344942957}  mat
  ▸ [2]                        pos={-6866.72510, -6739.01172, 352.239136} x={0.231891751, 0.750377536, -0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={-0.797913432, 0.260277271, 0.0344942957}  mat
  ▸ [3]                        pos={-6896.30566, -6721.00684, 412.660950} x={-0.621897459, 0.170435995, 0.538325846} y={0.256459028, 0.798714936, 0.0433964282} z={-0.503062546, 0.196484089, -0.643367350}  mat
  ▸ [4]                        pos={-6903.90430, -6718.92432, 419.238647} x={-0.795030117, 0.249507099, 0.106174320} y={0.256459028, 0.798714936, 0.0433964282} z={-0.0880657732, 0.0734890699, -0.832131743}  mat
  ▸ [5]                        pos={-6892.56396, -6721.99951, 408.815186} x={-0.576003432, 0.152776331, 0.592012644} y={0.256550431, 0.798679709, 0.0435033739} z={-0.554978788, 0.210641891, -0.594329834}  mat
  ▸ [6]                        pos={-6877.71094, -6774.56104, 366.353394} x={-0.231891751, -0.750377536, 0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={0.797913432, -0.260277271, -0.0344942957}  mat
  ▸ [7]                        pos={-6865.03955, -6733.55664, 350.073364} x={0.231891751, 0.750377536, -0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={-0.797913432, 0.260277271, 0.0344942957}  mat
  ▸ [8]                        pos={-6915.52539, -6715.27734, 420.790619} x={-0.745445311, 0.255176276, -0.291197509} y={0.256459028, 0.798714936, 0.0433964282} z={0.290068567, -0.0503935516, -0.786714792}  mat
  ▸ [9]                        pos={-6862.48145, -6725.27881, 346.786865} x={0.231891751, 0.750377536, -0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={-0.797913432, 0.260277271, 0.0344942957}  mat
  ▸ [10]                       pos={-6925.21631, -6711.95996, 417.005035} x={-0.674079120, 0.240337238, -0.439835101} y={0.256459028, 0.798714936, 0.0433964282} z={0.430634201, -0.0994608104, -0.714325726}  mat
  ▸ [11]                       pos={-6882.99365, -6791.65576, 373.140533} x={-0.231891751, -0.750377536, 0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={0.797913432, -0.260277271, -0.0344942957}  mat
  ▸ [12]                       pos={-6887.22559, -6805.36816, 378.716858} x={-0.231891751, -0.750377536, 0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={0.797913432, -0.260277271, -0.0344942957}  mat
  ▸ [13]                       pos={-6899.58838, -6845.37256, 394.600006} x={-0.231891751, -0.750377536, 0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={0.797913432, -0.260277271, -0.0344942957}  mat
▾ pSimulationWorldTransforms,14 0x00000000579603a0 {pos={-6904.11133, -6710.47217, 325.638428} x={0.772891462, -0.312591702, 0.102592200} y={0.313267827, 0.779267490, 0.0143335182} z={-0.100508697, 0.0250721127,  con
  ▾ [0]                        pos={-6904.11133, -6710.47217, 325.638428} x={0.772891462, -0.312591702, 0.102592200} y={0.313267827, 0.779267490, 0.0143335182} z={-0.100508697, 0.0250721127, 0.833588243}  con
    ▾ matrix3x4_t              pos={-6904.11133, -6710.47217, 325.638428} x={0.772891462, -0.312591702, 0.102592200} y={0.313267827, 0.779267490, 0.0143335182} z={-0.100508697, 0.0250721127, 0.833588243}  mat
      ▸ m_flMatVal             0x00000000579603a0 {0x00000000579603a0 {0.772891462, 0.313267827, -0.100508697, -6904.11133}, 0x00000000579603b0 {...}, ...}  float
  ▸ [1]                        pos={-6901.78857, -6711.84180, 349.343170} x={-0.0210168585, 0.00336329453, 0.839730203} y={0.333662868, 0.770870507, 0.00526344869} z={-0.770601928, 0.333687425, -0.0206231922}  con
  ▸ [2]                        pos={-6896.30273, -6697.04053, 322.380066} x={0.372540712, 0.314027518, -0.684251428} y={-0.294183373, 0.763462782, 0.190212324} z={0.693014741, 0.155278161, 0.448574603}  con
  ▸ [3]                        pos={-6907.72754, -6725.45947, 321.857361} x={-0.0869533047, 0.473728955, 0.688200593} y={0.656494319, 0.466711640, -0.238317832} z={-0.516772985, 0.513187230, -0.418550551}  con
  ▸ [4]                        pos={-6902.04639, -6711.83691, 364.934204} x={0.0402162559, -0.0126262503, 0.838941693} y={0.352931052, 0.762240112, -0.00544654671} z={-0.761197925, 0.352747172, 0.0417983755}  con
  ▸ [5]                        pos={-6900.82715, -6712.28955, 380.599396} x={0.0578526743, -0.140467137, 0.826148868} y={0.467432439, 0.692728996, 0.0850493684} z={-0.695528507, 0.453867257, 0.125875145}  con
  ▸ [6]                        pos={-6900.03711, -6714.20801, 391.881500} x={0.610547960, -0.455454558, 0.354107767} y={0.467432439, 0.692728996, 0.0850493684} z={-0.338139117, -0.135231882, 0.756950676}  con
  ▸ [7]                        pos={-6892.98975, -6711.08350, 381.304016} x={0.299724787, 0.722264886, 0.306754500} y={-0.648539186, 0.412867159, -0.338433862} z={-0.441771209, -0.116077729, 0.704956770}  con
  ▸ [8]                        pos={-6899.02979, -6720.03516, 380.205017} x={0.599534094, 0.587851763, 0.0242746510} y={0.587982833, -0.597418487, -0.0544741973} z={-0.0208577551, 0.0558717176, -0.837880313}  con
  ▸ [9]                        pos={-6890.17285, -6721.56641, 397.602539} x={0.798212171, -0.258422077, 0.0409316123} y={0.256249398, 0.798676491, 0.0452998802} z={-0.0528542995, -0.0305597484, 0.837778330}  con
  ▸ [10]                       pos={-6879.02051, -6716.74902, 393.418640} x={0.136251017, 0.478713423, -0.676660180} y={0.744388342, 0.230955645, 0.313281715} z={0.364584148, -0.650455952, -0.386762857}  con
  ▸ [11]                       pos={-6883.49463, -6730.69482, 392.627655} x={0.263435274, 0.442160517, 0.663849354} y={0.680697918, -0.489038348, 0.0556053631} z={0.415755033, 0.520514667, -0.511675775}  con
  ▸ [12]                       pos={-6892.62842, -6721.41406, 408.810089} x={-0.579993010, 0.152729258, 0.588117242} y={0.256249398, 0.798676491, 0.0452998802} z={-0.550948560, 0.210688487, -0.598051786}  con
  ▸ [13]                       pos={-6884.52002, -6723.33447, 396.799957} x={0.256185055, 0.798701406, 0.0452240407} y={-0.0526943989, -0.0305354241, 0.837789357} z={0.798243403, -0.258347869, 0.0407909080}  con
▾ this                         0x00000000112062e0 {m_DebugName={m_pName=0x0000000000000000 <NULL> m_nDebugIndex=187 } m_pWorld=0x000000002bdf700 {...} ...}  CRn
  ▸ m_DebugName                {m_pName=0x0000000000000000 <NULL> m_nDebugIndex=187 }  CRn
  ▸ m_pWorld                   0x000000002bdf700 {m_Filter={m_GroupPairs=0x000000002bdf700 {0x000000002bdf700 {1, 1, 0, 1, 1, 1, ...}, ...} } ...}  CRn
  ▸ m_pFeModel                 0x00000000110384f9 {m_nFlags=4009750271 m_nNodeCount=14 m_nCtrlCount=14 ...}  CFel
    m_flBendUnderRelax         0.000000000  float
    m_flStretchUnderRelax      0.000000000  float
    m_flOverPredict            0.000000000  float
    m_nNodeCount               14  unsi
    m_nParticleCount           14  unsi
  ▸ m_pParticles               0x0000000011266300 pos={-6876.29443, -6769.97656, 364.533264} x={-0.231891751, -0.750377536, 0.297926068} y={-0.123127304, -0.273476779, -0.784633875} z={0.797913432, -0.260277271,  mat
  ▸ m_pPos0                    0x0000000011266840 -1.#IND0000, -1.#IND0000, -1.#IND0000  Vect
  ▸ m_pPos1                    0x0000000011266920 -1.#IND0000, -1.#IND0000, -1.#IND0000  Vect
    m_nSimFlags                4  unsi
```

And the debugging tools we have are very basic: We have the Watch window, and printf. That's just inadequate.

Second Life of Testbed





☑ Shadows

5126.t=78.992

Dynamic Body #8, auto-sleeping on, sleeping, handle 0x00030002

Body Name:	Test Ragdoll/ValveBiped.Bip01_L_Hand		
Mass:	3.8		
Mass Priority	0		
Inertia:	20.5722961	32.7064362	42.3434029
Position:	17.6591110	152.9675140	67.3370667
Velocity:	1.8450111	-0.3921614	-273.1682129
Omega:	1.8860180	0.9076217	-0.1312281

☑ Continuous

Hull Shape 0 @ 0x5a6c640

```
V=16, E=74, F=23 centroid {2.6,-0.95,0.98}; 1.02 KB
proxy 15 box {13.3425,-8.78553,-13.2257}-{32.4748,10.2319,11.8269}
shape box {19.0848,-4.73048,-0.436229}-{27.9652,4.59435,6.97215}
Coll Group 26 "ragdoll", interact as 0x0000000000400000 with 0x0000000000000000 hier 0
```

□ More >>>

OrthoTwist Joint #4 of 14, handle=0xE0001

#8: Test Ragdoll/ValveBiped.Bip01_L_Hand

#7: Test Ragdoll/ValveBiped.Bip01_L_Forearm

Twists(XYZ) 0933533 0069960 6039

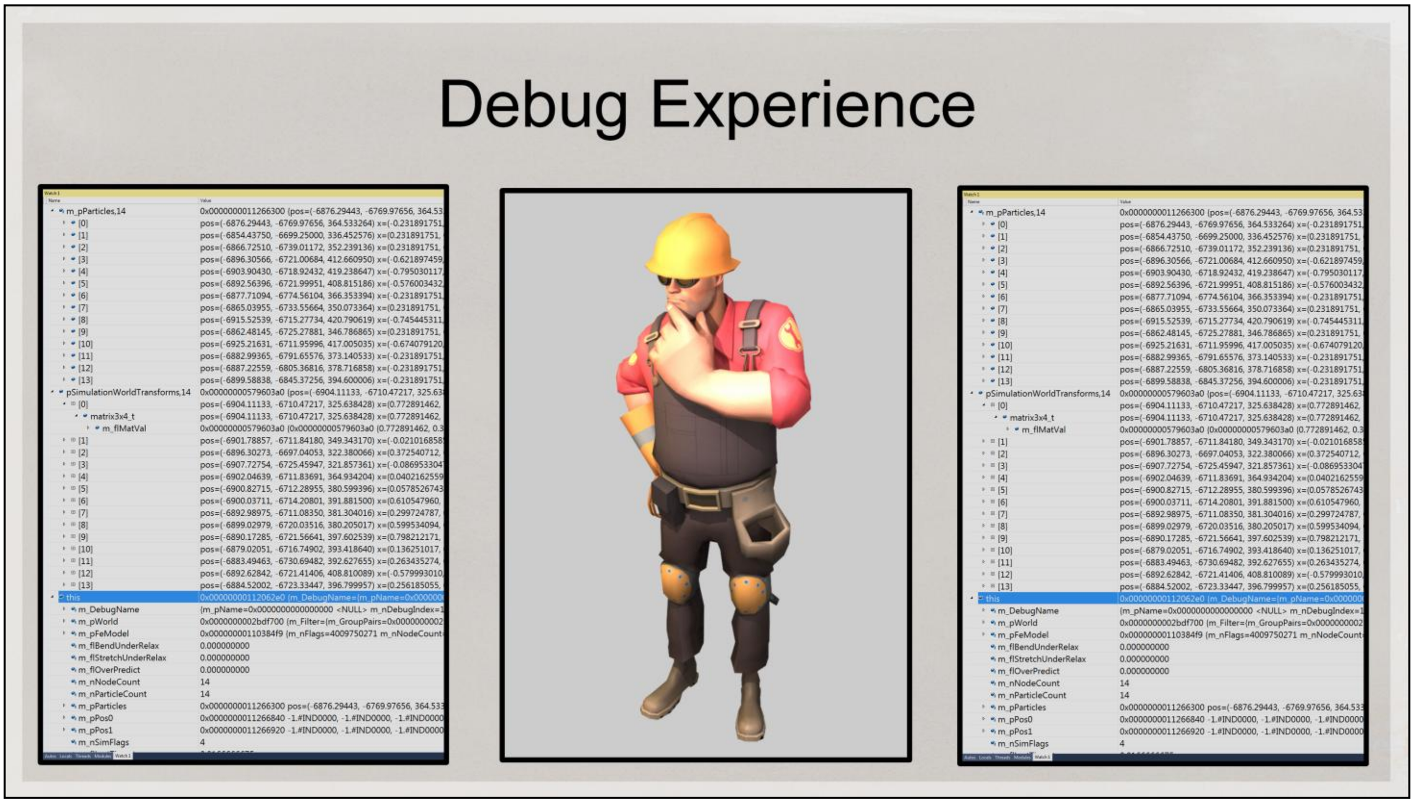
body #8[0]/16(16 asleep)

```
Broadphase {2008.0,2008.0,148.1} in. Vol ratio: 2.774 (0.460 leafs), SA ratio: 3.858 (0.936 leafs)
Tree height 5, 31 nodes, 16 leafs
```

Index	Name	Contacts: 4/4m. t0/0 pts
6	Test Ragdol...	1/1 0/0
7	Test Ragdol...	1/1 0/0
8	Test Ragdol...	1/1 0/0
9	Test Ragdol...	1/1 0/0

And now we have this wonderful application.. I wonder if we could use it when we step through the breakpoints just like we use it when we walk through the game.

We need to leverage it! But how?

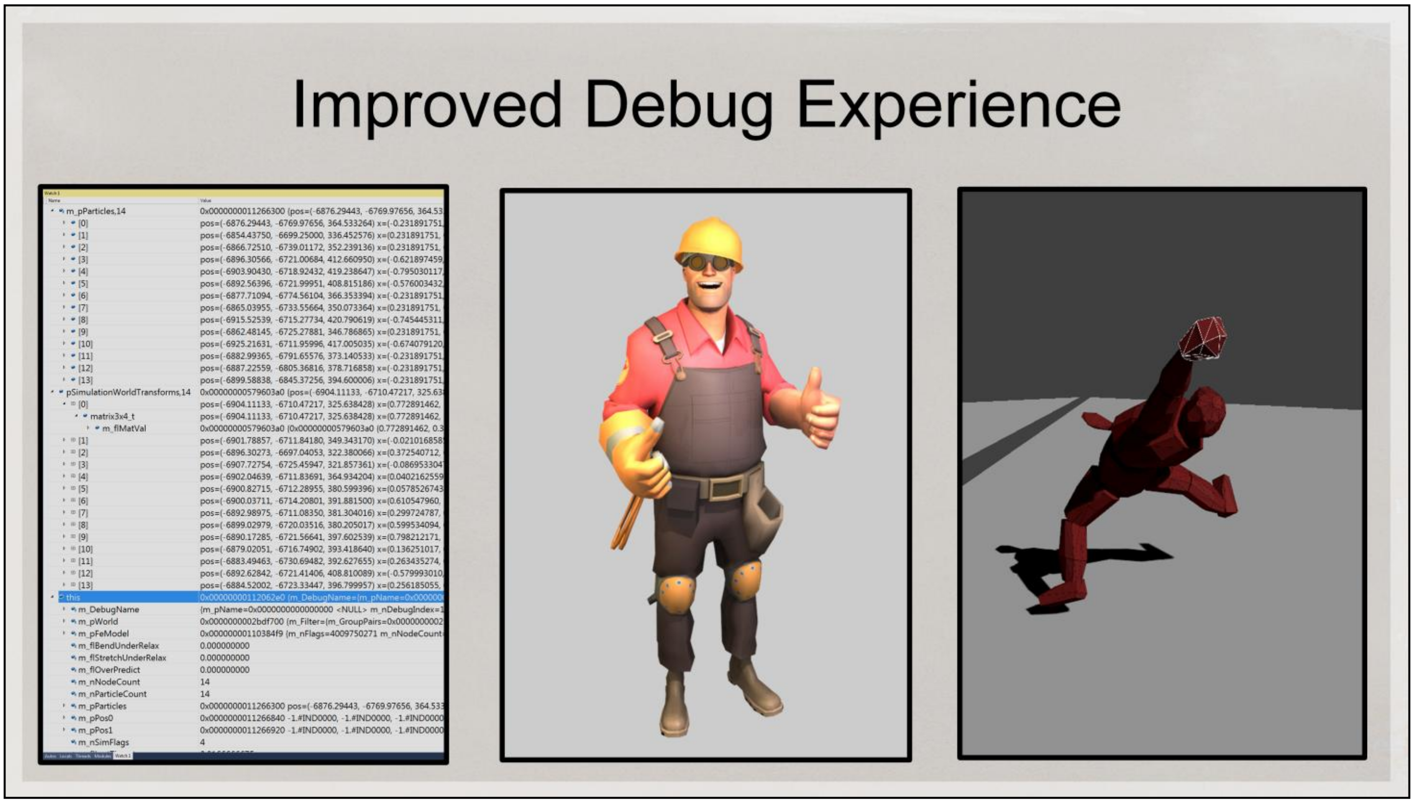


Debug Experience

```
Watch 1
Name                                   Value
m_pParticles,14                        0x0000000011266300 {pos=(-6876.29443, -6769.97656, 364.53
  [0]                                  pos=(-6876.29443, -6769.97656, 364.533264) x=(-0.231891751,
  [1]                                  pos=(-6854.43750, -6699.25000, 336.452576) x=(0.231891751,
  [2]                                  pos=(-6866.72510, -6739.01172, 352.239136) x=(0.231891751,
  [3]                                  pos=(-6896.30566, -6721.00684, 412.660950) x=(-0.621897459,
  [4]                                  pos=(-6903.90430, -6718.92432, 419.238647) x=(-0.795030117,
  [5]                                  pos=(-6892.56396, -6721.99951, 408.815186) x=(-0.576003432,
  [6]                                  pos=(-6877.71094, -6774.56104, 366.353394) x=(-0.231891751,
  [7]                                  pos=(-6865.03955, -6733.55664, 350.073364) x=(0.231891751,
  [8]                                  pos=(-6915.52539, -6715.27734, 420.790619) x=(-0.745445311,
  [9]                                  pos=(-6862.48145, -6725.27881, 346.786865) x=(0.231891751,
  [10]                                 pos=(-6925.21631, -6711.95996, 417.005035) x=(-0.674079120,
  [11]                                 pos=(-6882.99365, -6791.65576, 373.140533) x=(-0.231891751,
  [12]                                 pos=(-6887.22559, -6805.36816, 378.716858) x=(-0.231891751,
  [13]                                 pos=(-6899.58838, -6845.37256, 394.600006) x=(-0.231891751,
pSimulationWorldTransforms,14          0x00000000579603a0 {pos=(-6904.11133, -6710.47217, 325.63
  [0]                                  pos=(-6904.11133, -6710.47217, 325.638428) x=(0.772891462,
    matrix3x4_t                        pos=(-6904.11133, -6710.47217, 325.638428) x=(0.772891462,
      m_flMatVal                       0x00000000579603a0 {0x00000000579603a0 {0.772891462, 0.3
  [1]                                  pos=(-6901.78857, -6711.84180, 349.343170) x=(-0.021016858
  [2]                                  pos=(-6896.30273, -6697.04053, 322.380066) x=(0.372540712,
  [3]                                  pos=(-6907.72754, -6725.45947, 321.857361) x=(-0.086953304
  [4]                                  pos=(-6902.04639, -6711.83691, 364.934204) x=(0.0402162559
  [5]                                  pos=(-6900.82715, -6712.28955, 380.599396) x=(0.0578526743
  [6]                                  pos=(-6900.03711, -6714.20801, 391.881500) x=(0.610547960,
  [7]                                  pos=(-6892.98975, -6711.08350, 381.304016) x=(0.299724787,
  [8]                                  pos=(-6899.02979, -6720.03516, 380.205017) x=(0.599534094,
  [9]                                  pos=(-6890.17285, -6721.56641, 397.602539) x=(0.798212171,
  [10]                                 pos=(-6879.02051, -6716.74902, 393.418640) x=(0.136251017,
  [11]                                 pos=(-6883.49463, -6730.69482, 392.627655) x=(0.263435274,
  [12]                                 pos=(-6892.62842, -6721.41406, 408.810089) x=(-0.579993010,
  [13]                                 pos=(-6884.52002, -6723.33447, 396.799957) x=(0.256185055,
this                                   0x00000000112062e0 {m_DebugName={m_pName=0x000000
  m_DebugName                          {m_pName=0x0000000000000000 <NULL> m_nDebugIndex=1
  m_pWorld                             0x0000000002bdf700 {m_Filter={m_GroupPairs=0x0000000002
  m_pFeModel                           0x00000000110384f9 {m_nFlags=4009750271 m_nNodeCount=
  m_flBendUnderRelax                   0.000000000
  m_flStretchUnderRelax                0.000000000
  m_flOverPredict                      0.000000000
  m_nNodeCount                         14
  m_nParticleCount                     14
  m_pParticles                         0x0000000011266300 pos=(-6876.29443, -6769.97656, 364.533
  m_pPos0                              0x0000000011266840 -1.#IND0000, -1.#IND0000, -1.#IND0000
  m_pPos1                              0x0000000011266920 -1.#IND0000, -1.#IND0000, -1.#IND0000
  m_nSimFlags                          4
```

```
Watch 1
Name                                   Value
m_pParticles,14                        0x0000000011266300 {pos=(-6876.29443, -6769.97656, 364.53
  [0]                                  pos=(-6876.29443, -6769.97656, 364.533264) x=(-0.231891751,
  [1]                                  pos=(-6854.43750, -6699.25000, 336.452576) x=(0.231891751,
  [2]                                  pos=(-6866.72510, -6739.01172, 352.239136) x=(0.231891751,
  [3]                                  pos=(-6896.30566, -6721.00684, 412.660950) x=(-0.621897459,
  [4]                                  pos=(-6903.90430, -6718.92432, 419.238647) x=(-0.795030117,
  [5]                                  pos=(-6892.56396, -6721.99951, 408.815186) x=(-0.576003432,
  [6]                                  pos=(-6877.71094, -6774.56104, 366.353394) x=(-0.231891751,
  [7]                                  pos=(-6865.03955, -6733.55664, 350.073364) x=(0.231891751,
  [8]                                  pos=(-6915.52539, -6715.27734, 420.790619) x=(-0.745445311,
  [9]                                  pos=(-6862.48145, -6725.27881, 346.786865) x=(0.231891751,
  [10]                                 pos=(-6925.21631, -6711.95996, 417.005035) x=(-0.674079120,
  [11]                                 pos=(-6882.99365, -6791.65576, 373.140533) x=(-0.231891751,
  [12]                                 pos=(-6887.22559, -6805.36816, 378.716858) x=(-0.231891751,
  [13]                                 pos=(-6899.58838, -6845.37256, 394.600006) x=(-0.231891751,
pSimulationWorldTransforms,14          0x00000000579603a0 {pos=(-6904.11133, -6710.47217, 325.63
  [0]                                  pos=(-6904.11133, -6710.47217, 325.638428) x=(0.772891462,
    matrix3x4_t                        pos=(-6904.11133, -6710.47217, 325.638428) x=(0.772891462,
      m_flMatVal                       0x00000000579603a0 {0x00000000579603a0 {0.772891462, 0.3
  [1]                                  pos=(-6901.78857, -6711.84180, 349.343170) x=(-0.021016858
  [2]                                  pos=(-6896.30273, -6697.04053, 322.380066) x=(0.372540712,
  [3]                                  pos=(-6907.72754, -6725.45947, 321.857361) x=(-0.086953304
  [4]                                  pos=(-6902.04639, -6711.83691, 364.934204) x=(0.0402162559
  [5]                                  pos=(-6900.82715, -6712.28955, 380.599396) x=(0.0578526743
  [6]                                  pos=(-6900.03711, -6714.20801, 391.881500) x=(0.610547960,
  [7]                                  pos=(-6892.98975, -6711.08350, 381.304016) x=(0.299724787,
  [8]                                  pos=(-6899.02979, -6720.03516, 380.205017) x=(0.599534094,
  [9]                                  pos=(-6890.17285, -6721.56641, 397.602539) x=(0.798212171,
  [10]                                 pos=(-6879.02051, -6716.74902, 393.418640) x=(0.136251017,
  [11]                                 pos=(-6883.49463, -6730.69482, 392.627655) x=(0.263435274,
  [12]                                 pos=(-6892.62842, -6721.41406, 408.810089) x=(-0.579993010,
  [13]                                 pos=(-6884.52002, -6723.33447, 396.799957) x=(0.256185055,
this                                   0x00000000112062e0 {m_DebugName={m_pName=0x000000
  m_DebugName                          {m_pName=0x0000000000000000 <NULL> m_nDebugIndex=1
  m_pWorld                             0x0000000002bdf700 {m_Filter={m_GroupPairs=0x0000000002
  m_pFeModel                           0x00000000110384f9 {m_nFlags=4009750271 m_nNodeCount=
  m_flBendUnderRelax                   0.000000000
  m_flStretchUnderRelax                0.000000000
  m_flOverPredict                      0.000000000
  m_nNodeCount                         14
  m_nParticleCount                     14
  m_pParticles                         0x0000000011266300 pos=(-6876.29443, -6769.97656, 364.533
  m_pPos0                              0x0000000011266840 -1.#IND0000, -1.#IND0000, -1.#IND0000
  m_pPos1                              0x0000000011266920 -1.#IND0000, -1.#IND0000, -1.#IND0000
  m_nSimFlags                          4
```

The usual debug experience consists of sifting through Watch windows, stepping through lines of code, imagining our 3D world in our mind's eye..



Improved Debug Experience

```
Name                              Value
m_pParticles,14                   0x0000000011266300 {pos=(-6876.29443, -6769.97656, 364.53
  [0]                             pos=(-6876.29443, -6769.97656, 364.533264) x=(-0.231891751
  [1]                             pos=(-6854.43750, -6699.25000, 336.452576) x=(0.231891751,
  [2]                             pos=(-6866.72510, -6739.01172, 352.239136) x=(0.231891751,
  [3]                             pos=(-6896.30566, -6721.00684, 412.660950) x=(-0.621897459
  [4]                             pos=(-6903.90430, -6718.92432, 419.238647) x=(-0.795030117
  [5]                             pos=(-6892.56396, -6721.99951, 408.815186) x=(-0.576003432
  [6]                             pos=(-6877.71094, -6774.56104, 366.353394) x=(-0.231891751
  [7]                             pos=(-6865.03955, -6733.55664, 350.073364) x=(0.231891751,
  [8]                             pos=(-6915.52539, -6715.27734, 420.790619) x=(-0.745445311
  [9]                             pos=(-6862.48145, -6725.27881, 346.786865) x=(0.231891751,
  [10]                            pos=(-6925.21631, -6711.95996, 417.005035) x=(-0.674079120
  [11]                            pos=(-6882.99365, -6791.65576, 373.140533) x=(-0.231891751
  [12]                            pos=(-6887.22559, -6805.36816, 378.716858) x=(-0.231891751
  [13]                            pos=(-6899.58838, -6845.37256, 394.600006) x=(-0.231891751
pSimulationWorldTransforms,14     0x00000000579603a0 {pos=(-6904.11133, -6710.47217, 325.63
  [0]                             pos=(-6904.11133, -6710.47217, 325.638428) x=(0.772891462,
    matrix3x4_t                   pos=(-6904.11133, -6710.47217, 325.638428) x=(0.772891462,
      m_flMatVal                  0x00000000579603a0 {0x00000000579603a0 {0.772891462, 0.3
  [1]                             pos=(-6901.78857, -6711.84180, 349.343170) x=(-0.021016858
  [2]                             pos=(-6896.30273, -6697.04053, 322.380066) x=(0.372540712,
  [3]                             pos=(-6907.72754, -6725.45947, 321.857361) x=(-0.086953304
  [4]                             pos=(-6902.04639, -6711.83691, 364.934204) x=(0.0402162559
  [5]                             pos=(-6900.82715, -6712.28955, 380.599396) x=(0.0578526743
  [6]                             pos=(-6900.03711, -6714.20801, 391.881500) x=(0.610547960,
  [7]                             pos=(-6892.98975, -6711.08350, 381.304016) x=(0.299724787,
  [8]                             pos=(-6899.02979, -6720.03516, 380.205017) x=(0.599534094,
  [9]                             pos=(-6890.17285, -6721.56641, 397.602539) x=(0.798212171,
  [10]                            pos=(-6879.02051, -6716.74902, 393.418640) x=(0.136251017,
  [11]                            pos=(-6883.49463, -6730.69482, 392.627655) x=(0.263435274,
  [12]                            pos=(-6892.62842, -6721.41406, 408.810089) x=(-0.579993010
  [13]                            pos=(-6884.52002, -6723.33447, 396.799957) x=(0.256185055,
this                              0x00000000112062e0 {m_DebugName={m_pName=0x000000
  m_DebugName                     {m_pName=0x0000000000000000 <NULL> m_nDebugIndex=1
  m_pWorld                        0x000000002bdf700 {m_Filter={m_GroupPairs=0x000000002
  m_pFeModel                      0x00000000110384f9 {m_nFlags=4009750271 m_nNodeCount
  m_flBendUnderRelax              0.000000000
  m_flStretchUnderRelax           0.000000000
  m_flOverPredict                 0.000000000
  m_nNodeCount                    14
  m_nParticleCount                14
  m_pParticles                    0x0000000011266300 pos=(-6876.29443, -6769.97656, 364.533
  m_pPos0                         0x0000000011266840 -1.#IND0000, -1.#IND0000, -1.#IND0000
  m_pPos1                         0x0000000011266920 -1.#IND0000, -1.#IND0000, -1.#IND0000
  m_nSimFlags                     4
```

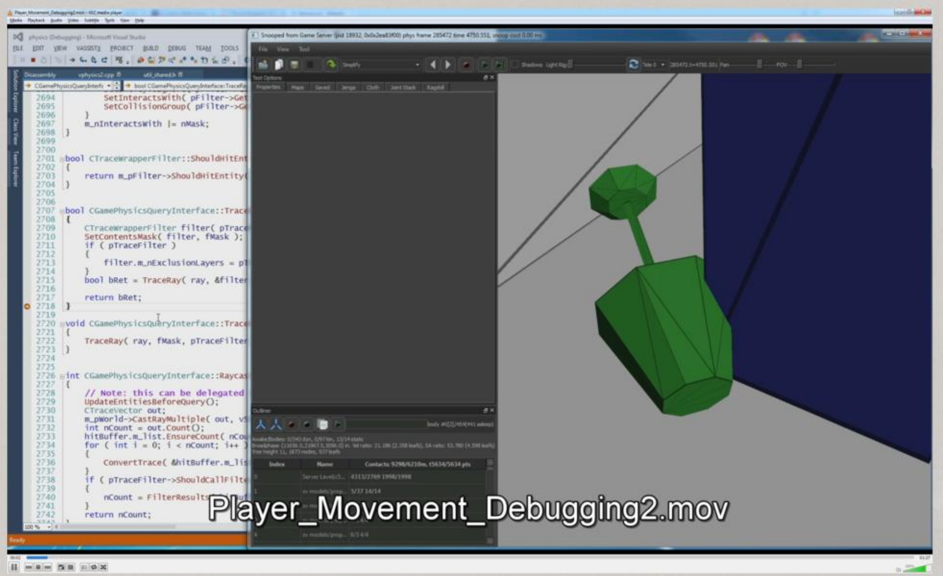
Ideally, I want to see the world just like I did in the Physics Testbed, but I also want to see it changing just in time as I step through my breakpoints.

I want to be able to fly around the physics world frozen at a breakpoint. That would be ideal for me.

That would make debugging much more comfortable, enjoyable and productive.

Ideally, I want to see the world just like I did in the Physics Testbed, but I also want to see it changing just in time as I step through my breakpoints.
I want to be able to fly around the physics world frozen at a breakpoint. That would be ideal for me.
That would make debugging much more comfortable, enjoyable and productive.

Debugging Visually



```
            SetInteractsWith( pFilter->Get
            SetCollisionGroup( pFilter->Ge
        }
        m_nInteractsWith |= nMask;
}


bool CTraceWrapperFilter::ShouldHitEnt
{
        return m_pFilter->ShouldHitEntity(
}


bool CGamePhysicsQueryInterface::Trace
{
        CTraceWrapperFilter filter( pTrace
        SetContentsMask( filter, fMask );
        if ( pTraceFilter )
        {
            filter.m_nExclusionLayers = pT
        }
        bool bRet = TraceRay( ray, &filter

        return bRet;
}

void CGamePhysicsQueryInterface::Trace
{
        TraceRay( ray, fMask, pTraceFilter
}


int CGamePhysicsQueryInterface::Raycas
{
        // Note: this can be delegated
        UpdateEntitiesBeforeQuery();
        CTraceVector out;
        m_pWorld->CastRayMultiple( out, vS
        int nCount = out.Count();
        hitBuffer.m_list.EnsureCount( nCou
        for ( int i = 0; i < nCount; i++ )
        {
            ConvertTrace( &hitBuffer.m_lis
        }
        if ( pTraceFilter->ShouldCallFilte
        {
            nCount = FilterResults
        }
        return nCount;
```

Player_Movement_Debugging2.mov

Let me show you what I mean.

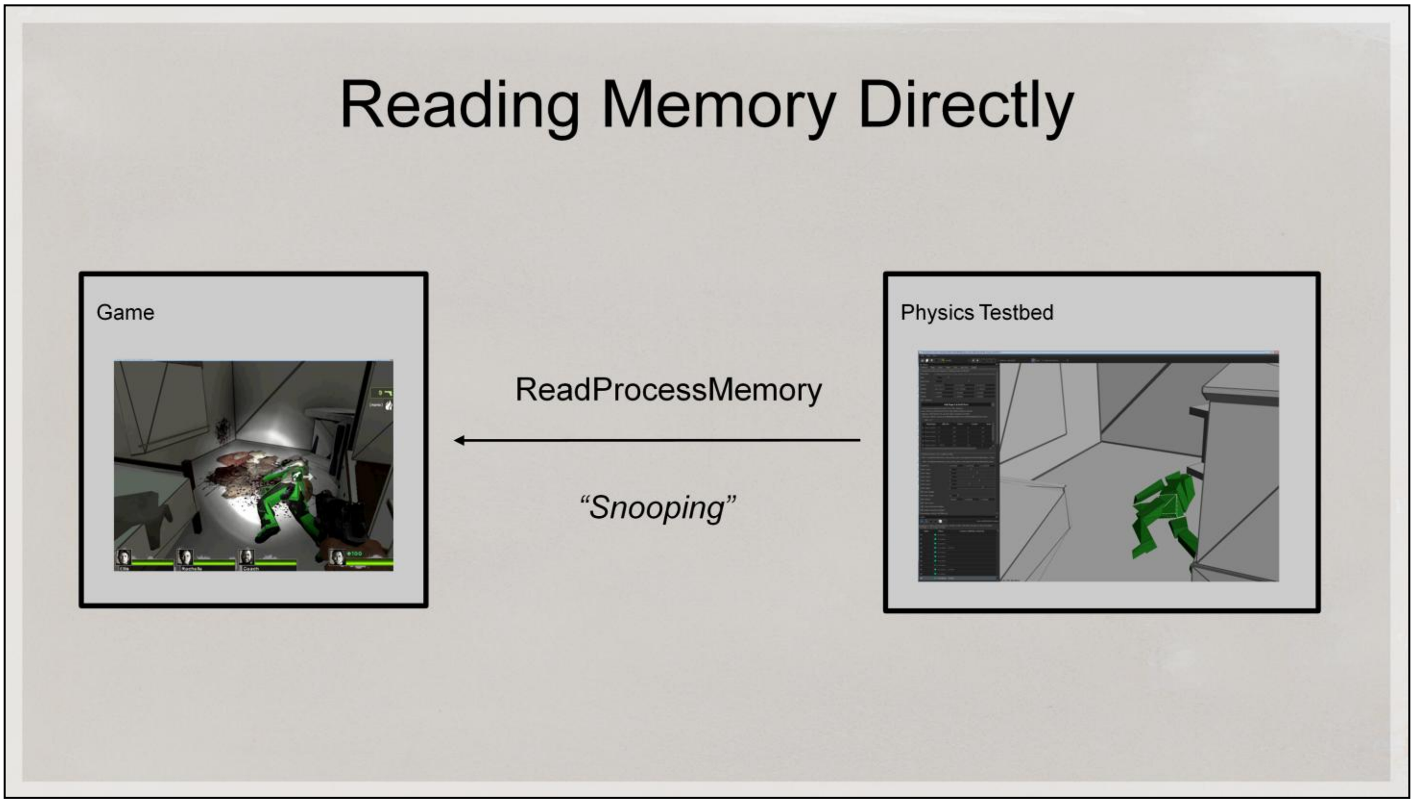
As you can see, this is the place with too many box casts that I discovered in the previous demo.

I place a breakpoint in the box trace function. Every time the it hits, I'm looking at the 3D world. I'm seeing the very last box cast.

This lets me step through bunch of casts, visualizing each of them in Physics Testbed.

This whole thing happens inside of a frame.

Please note that the game didn't send any data packets to Physics Testbed. It's casting boxes in a very tight loop. The testbed is effectively acting like a watch window.



Reading Memory Directly

Game

ReadProcessMemory

“Snooping”

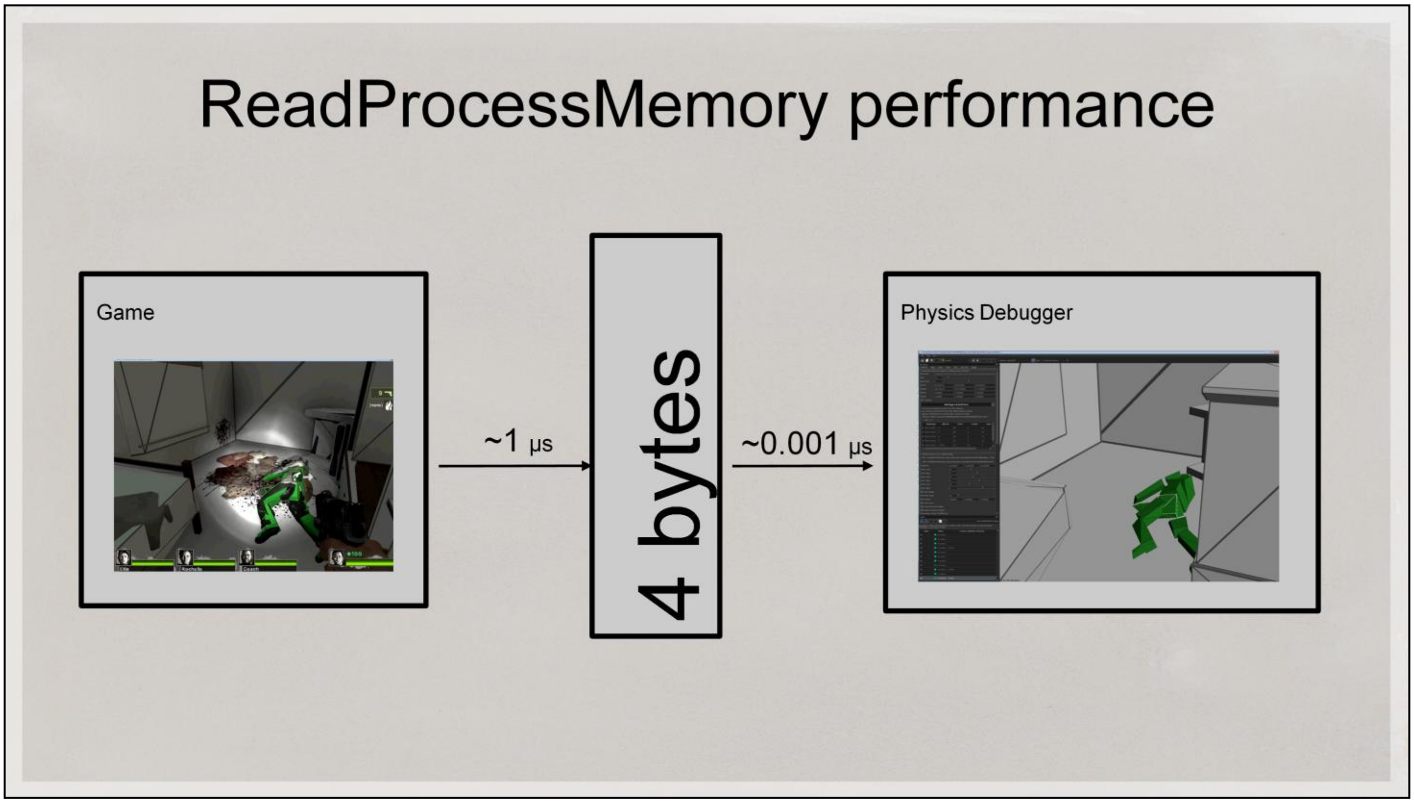
Physics Testbed

The method that lets us do it is ReadProcessMemory API.

The Physics Testbed, acting in a way like a debugger, reads memory from the game without the game knowing it. I call this snooping. There is no way for the debugger to alter the game state, which is a nice benefit.

It works when process is stopped in debugger

There is no activity required from the game side besides a mutex and advertising where to start the search (the root data structure). Both are trivial to implement and are not intrusive.



ReadProcessMemory performance

Game

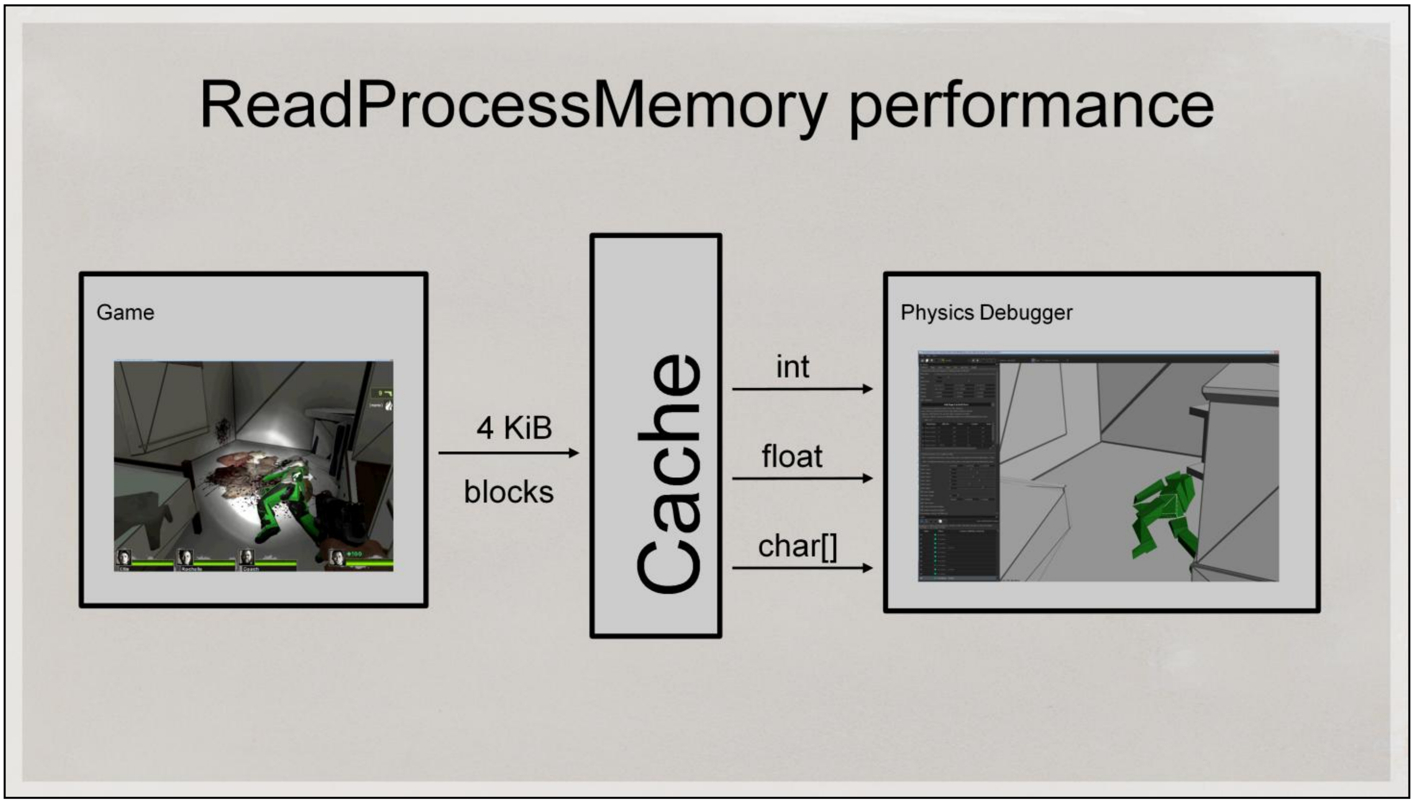
~1 µs

4 bytes

~0.001 µs

Physics Debugger

- The biggest question when I thought of this idea was whether it's fast enough. So I wrote a small benchmark.
- It is slow to read every 4- or 8-byte field across the Process boundary
- Every call to ReadProcessMemory descends into kernel, performs a syscall — it costs you at least 1us, limiting your bandwidth.
- But just copying 4 bytes in-process takes a thousandth of that time.



ReadProcessMemory performance

Game

4 KiB

blocks

Cache

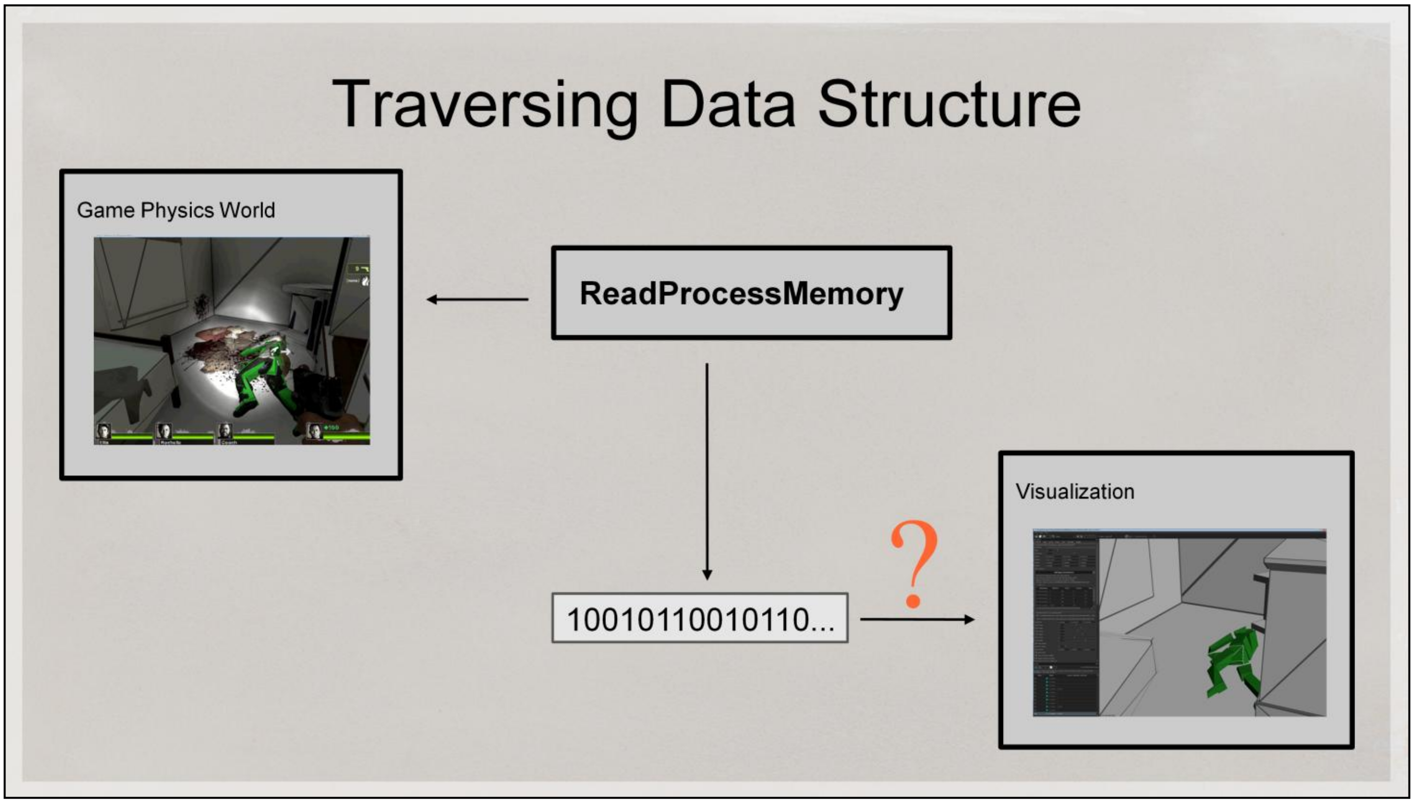
int

float

char[]

Physics Debugger

- I simply implemented software cache layer that reads at least 4Kb page every time you snoop a byte and caches that page
- You always know you can read the 4KiB page if you can read a single byte of it. There will be no memory protection faults.
- This cache helps a lot with reading C NULL-terminated strings, for which length is unknown until you read the whole string one byte at a time.



Traversing Data Structure

Game Physics World

ReadProcessMemory

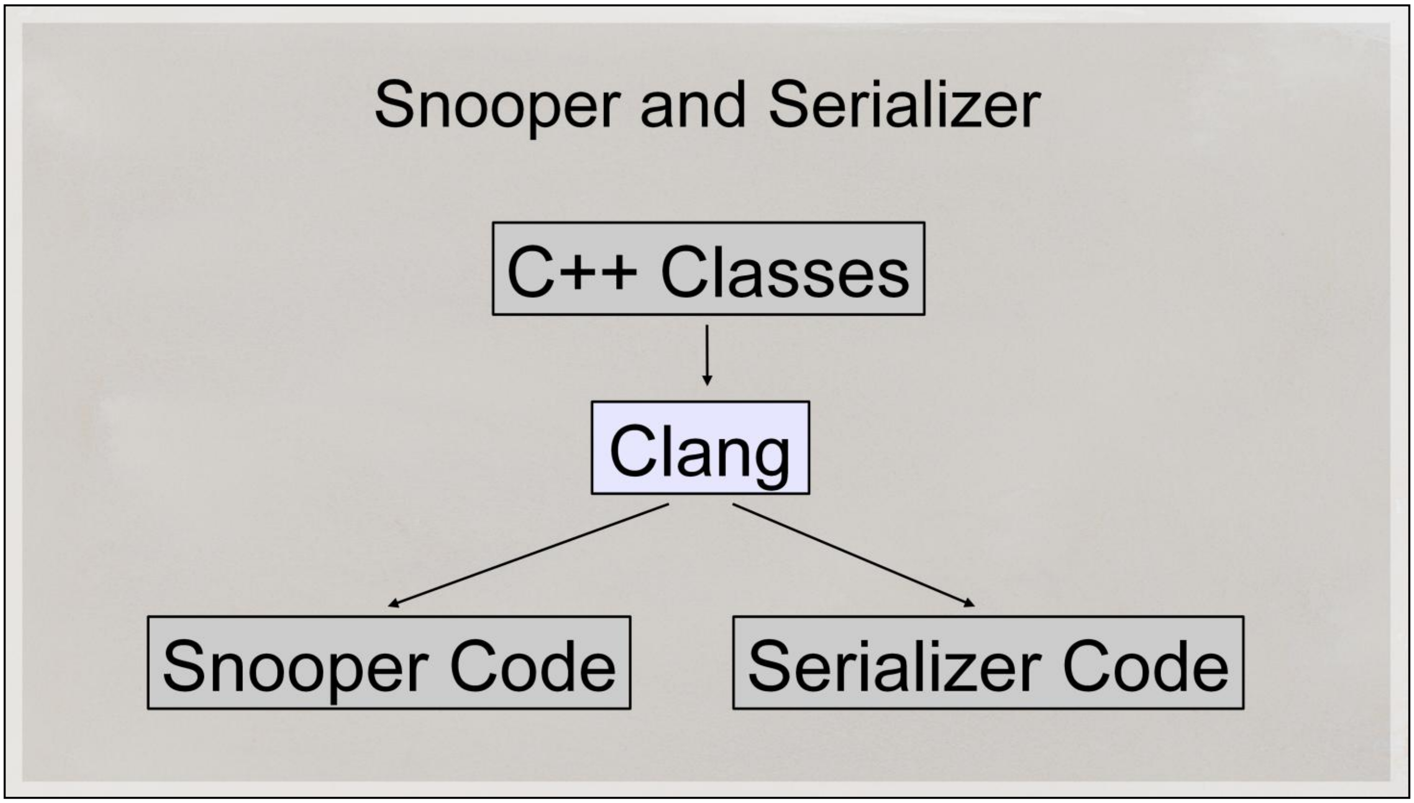
10010110010110...

?

Visualization

So, we can snoop the bytes from the game physics, but how do we know what to do with them?

In fact, walking the memory of the other process is very much like serialization. If you know your data structures, you can traverse them in another process using ReadProcessMemory just the same as you traverse them in the same process when you serialize them.



Snooper and Serializer

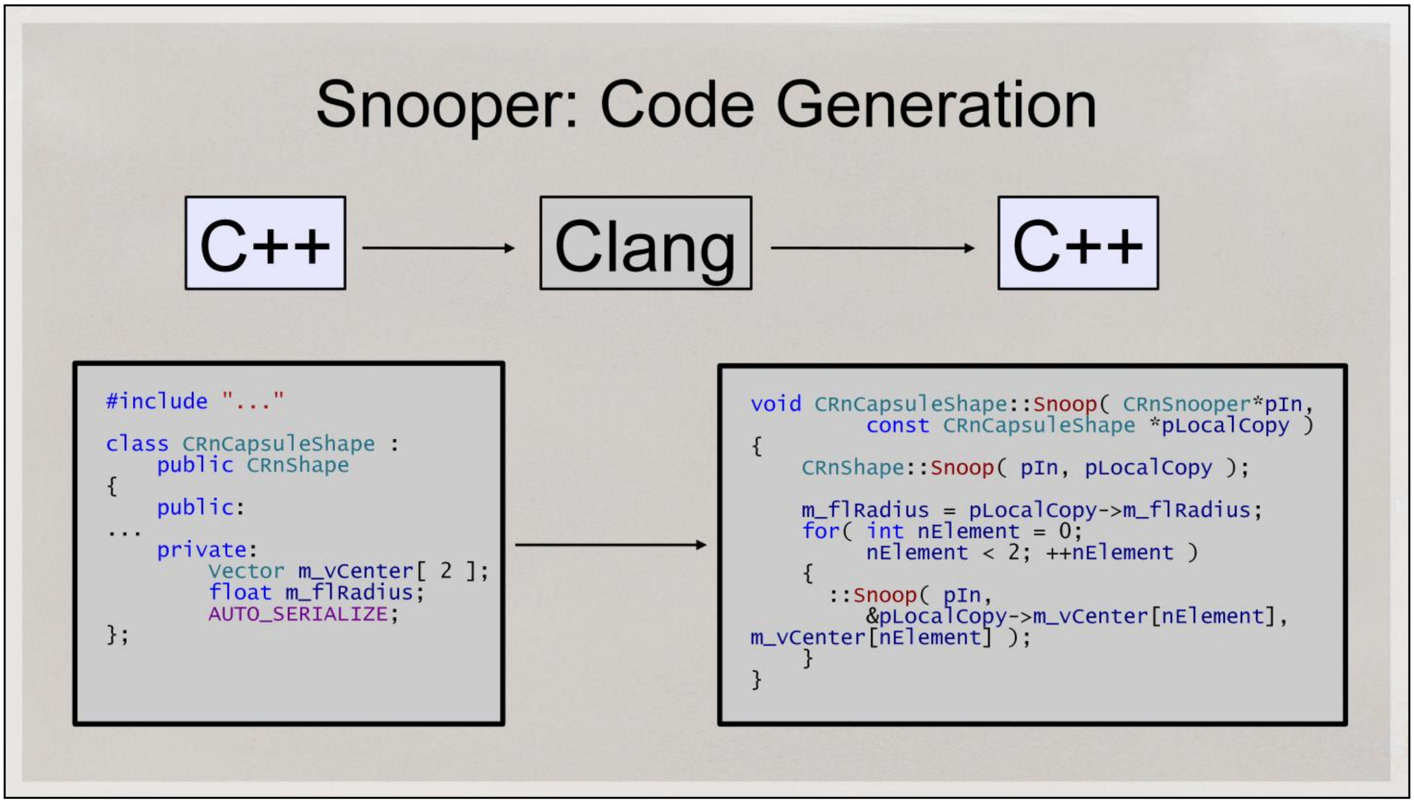
C++ Classes

Clang

Snooper Code

Serializer Code

- We can still use Clang to understand our data structures.
- We can use all our existing code as input and spit out the snooper routines.

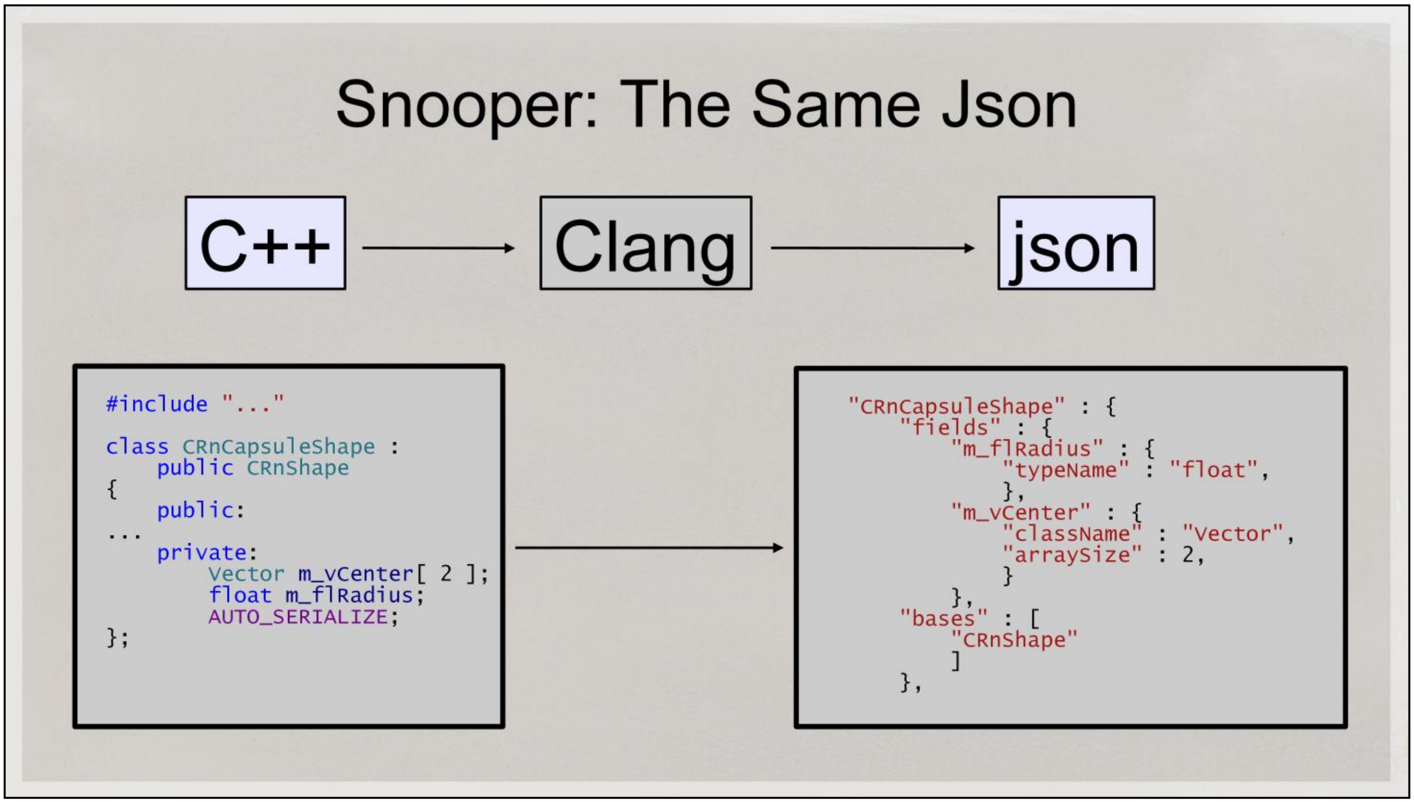


Snooper: Code Generation

```
#include "..."

class CRnCapsuleShape :
    public CRnShape
{
    public:
...
    private:
        Vector m_vCenter[ 2 ];
        float m_flRadius;
        AUTO_SERIALIZE;
};
```

```
void CRnCapsuleShape::Snoop( CRnSnooper*pIn,
         const CRnCapsuleShape *pLocalCopy )
{
    CRnShape::Snoop( pIn, pLocalCopy );

    m_flRadius = pLocalCopy->m_flRadius;
    for( int nElement = 0;
         nElement < 2; ++nElement )
    {
      ::Snoop( pIn,
         &pLocalCopy->m_vCenter[nElement],
m_vCenter[nElement] );
    }
}
```

- Just like with the serializer, we are transforming C++ code into additional C++ code for snooper
- Something is becoming clearer now. Generating code as a bunch of printf() statements would really be awkward. Because we now have to print out serialize, unserialize, snoop and possibly collect statistics routines.

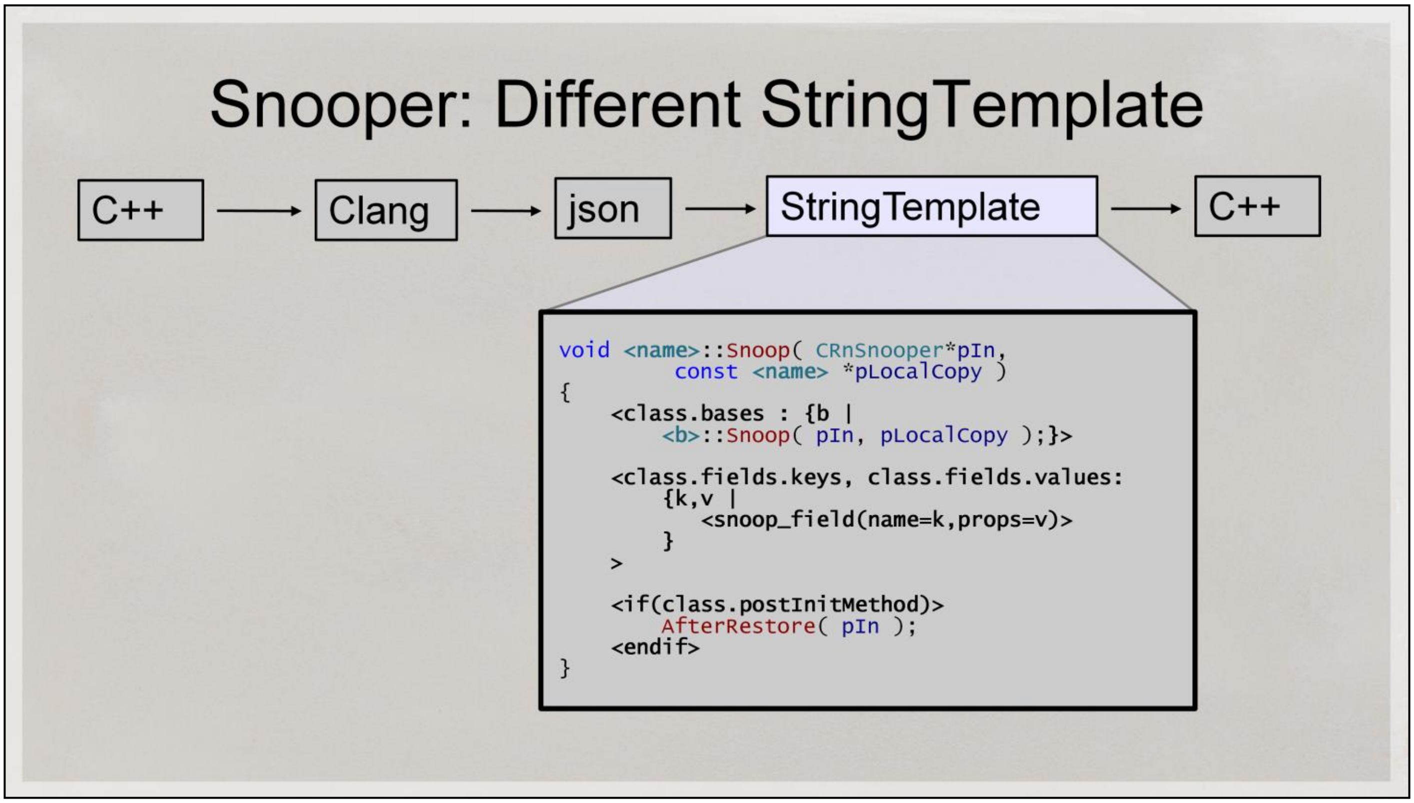


Snooper: The Same Json

```
#include "..."

class CRnCapsuleShape :
    public CRnShape
{
    public:
...
    private:
        Vector m_vCenter[ 2 ];
        float m_flRadius;
        AUTO_SERIALIZE;
};
```

```
"CRnCapsuleShape" : {
    "fields" : {
        "m_flRadius" : {
            "typeName" : "float",
            },
        "m_vCenter" : {
            "className" : "Vector",
            "arraySize" : 2,
            }
        },
    "bases" : [
        "CRnShape"
        ]
    },
```

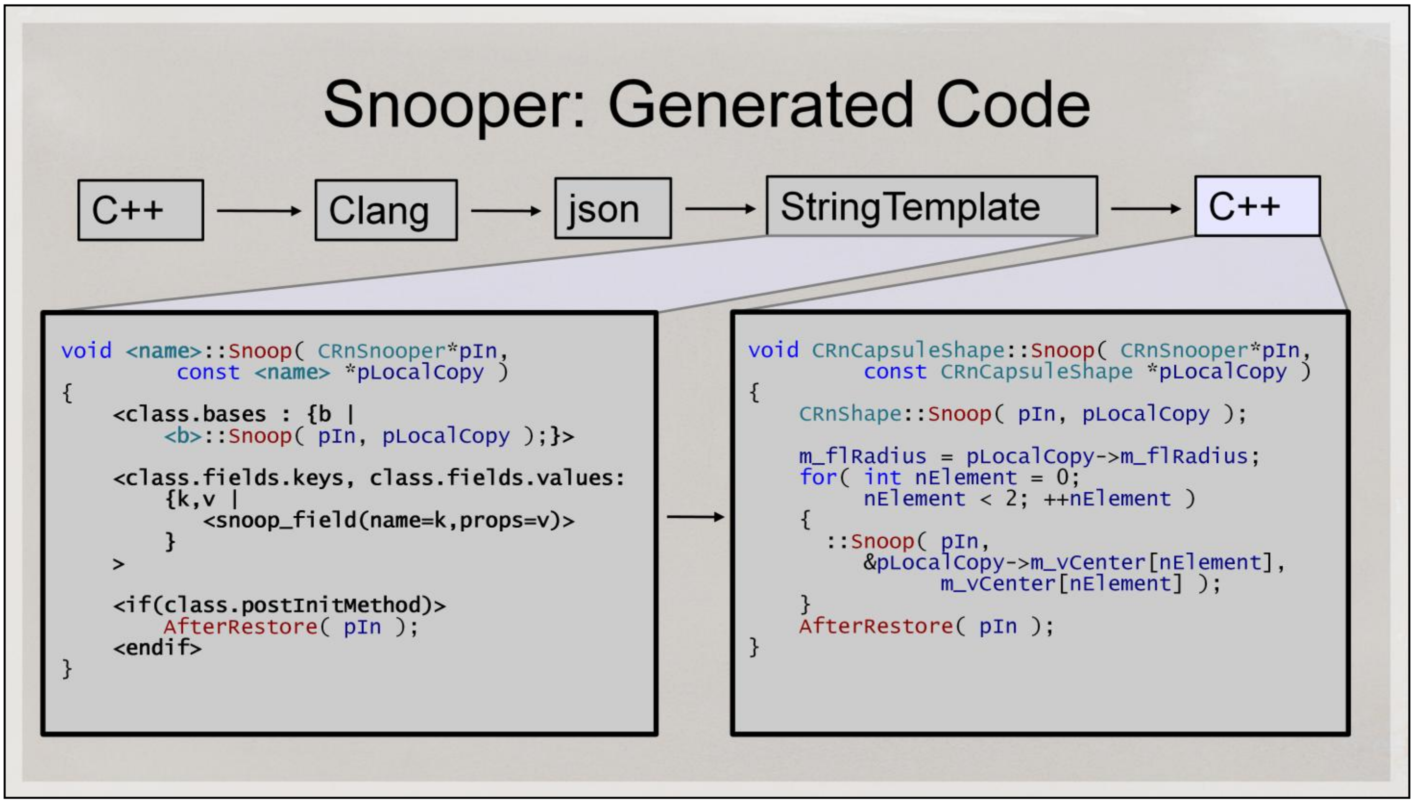
So, we still generate the same json file as we did for serialization.



Snooper: Different StringTemplate

```
void <name>::Snoop( CRnSnooper*pIn,
          const <name> *pLocalCopy )
{
    <class.bases : {b |
        <b>::Snoop( pIn, pLocalCopy );}>

    <class.fields.keys, class.fields.values:
        {k,v |
           <snoop_field(name=k,props=v)>
        }
    >

    <if(class.postInitMethod)>
        AfterRestore( pIn );
    <endif>
}
```

- The StringTemplate for the snooper looks like this. It's a different string template, but it's small. All my snoop code is generated from a 200-line template. These are a few of those lines. They auto-generate the code to snoop most classes in our physics engine.
- You feed it the same json file as for the serializer. It spits out snooper C++ file. It's that simple.



Snooper: Generated Code

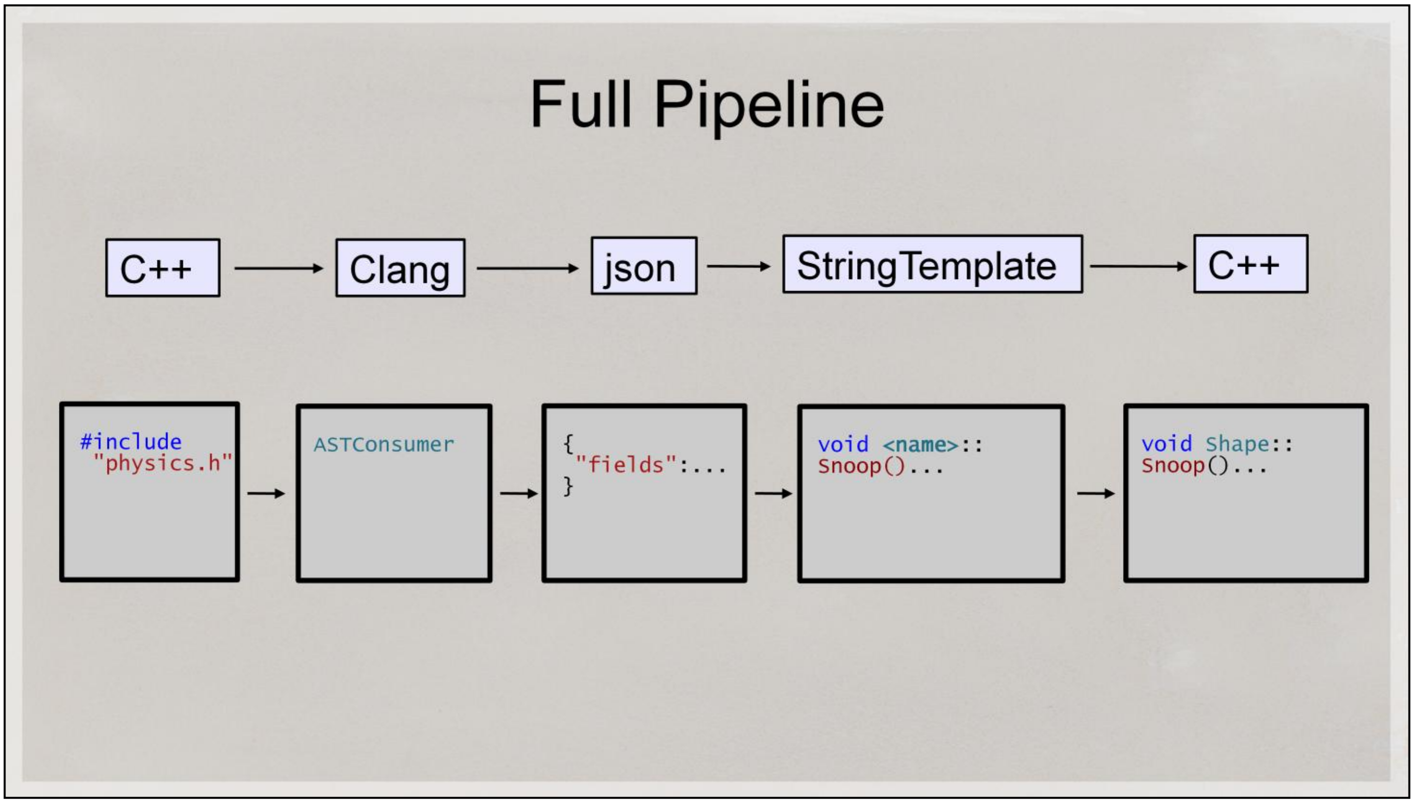
C++ → Clang → json → StringTemplate → C++

```
void <name>::Snoop( CRnSnooper*pIn,
         const <name> *pLocalCopy )
{
    <class.bases : {b |
        <b>::Snoop( pIn, pLocalCopy );}>

    <class.fields.keys, class.fields.values:
        {k,v |
           <snoop_field(name=k,props=v)>
        }
    >

    <if(class.postInitMethod)>
        AfterRestore( pIn );
    <endif>
}
```

→

```
void CRnCapsuleShape::Snoop( CRnSnooper*pIn,
         const CRnCapsuleShape *pLocalCopy )
{
    CRnShape::Snoop( pIn, pLocalCopy );

    m_flRadius = pLocalCopy->m_flRadius;
    for( int nElement = 0;
         nElement < 2; ++nElement )
    {
      ::Snoop( pIn,
         &pLocalCopy->m_vCenter[nElement],
               m_vCenter[nElement] );
    }
    AfterRestore( pIn );
}
```

- This here is the template and the resulting generated C++ code for snooper.
- I'm snooping the whole struct into a local copy. Then I read out all the plain data elements.
- All the non-plain data elements are recursively snooped by other overloaded Snoop functions.
- The template is smart enough to know where I have arrays, templates, pointers to follow and so on.



Full Pipeline

C++ → Clang → json → StringTemplate → C++

```
#include
 "physics.h"
```

```
ASTConsumer
```

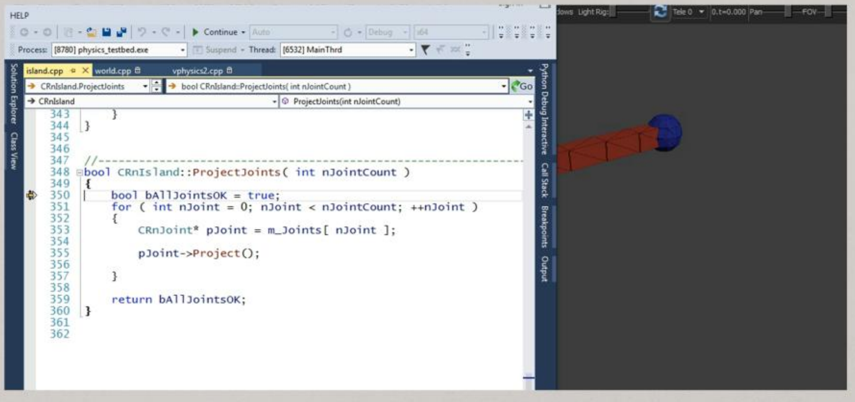
```
{
 "fields":...
}
```

```
void <name>::
Snoop()...
```

```
void Shape::
Snoop()...
```

- Just to bring the point home, here's how the full code parsing and generation pipeline looks in our case
- Parsing C++ is probably the most intimidating thing in this case. But after you spend a day or two learning Clang's API, you might find it rather intuitive.

Example: Snooping Joint Stack



```
    }
}


//-----------------------------------------------------------
bool CRnIsland::ProjectJoints( int nJointCount )
{
    bool bAllJointsOK = true;
    for ( int nJoint = 0; nJoint < nJointCount; ++nJoint )
    {
        CRnJoint* pJoint = m_Joints[ nJoint ];

        pJoint->Project();

    }

    return bAllJointsOK;
}
```

Let's see a little demo of what we have as a result.

I configured a projection-type solver of a chain of hinge joints. As you can see, small chains solve fine, but larger chains start to break constraints.

I wanted to see how the solver behaves inside the iteration loop.

So I put a breakpoint where each joint is solved.

After each breakpoint hits, I switch to the Testbed and snoop. I have two testbeds here, one debugging the other. Notice that one is unresponsive – that's the one that I'm debugging. And another one acts like a watch window.

So what we have here effectively a visualization of the internal loop of a projection constraint solver.

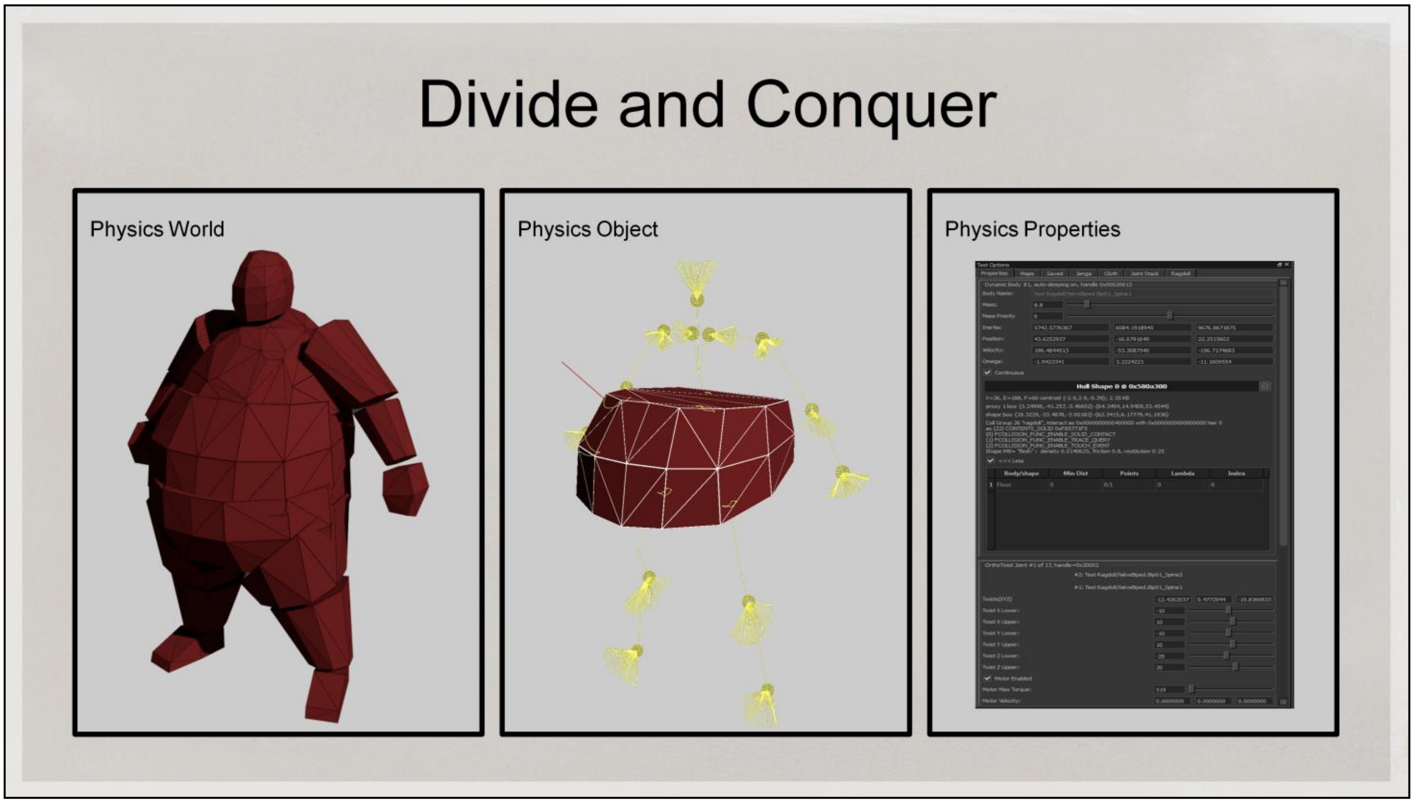
Details

Fixing stuff, whether it's yours or not, is unavoidable. At least if you make anything complex.

The faster you can drill down to details, the more efficient you are while debugging problems, the more time you spend developing and not fixing stuff. Once you make debugging 10x faster and more convenient, there's a quantum leap : you can develop features with complexity that would otherwise be infeasible to develop.

That's why it's important to make the process as seamless and convenient as possible. It pays with higher productivity in the end.

There are a few more technical details, though



Divide and Conquer

Physics World

Physics Object

Physics Properties

```
Test Options
Properties | Maps | Saved | Jenga | Cloth | Joint Stack | Ragdoll
Dynamic Body #1, auto-sleeping on, handle 0x00020013
Body Name:     Test Ragdoll/ValveBiped.Bip01_Spine1
Mass:          8.8
Mass Priority: 0
Inertia:       5742.5776367   6084.1918945   9676.8671875
Position:      43.6252937     -16.6791649    22.2510602
Velocity:      196.4844513    -53.3087540    -156.7174683
Omega:         -1.9423341     3.2224221      -11.1609554
✓ Continuous
                    Hull Shape 0 @ 0x580a300
V=36, E=188, F=60 centroid {-2.9,2.9,-0.29}; 2.35 KB
proxy 1 box {3.24996,-41.257,-3.46602}-{64.3494,14.9409,53.4544}
shape box {18.3229,-33.4878,-3.00183}-{62.3415,6.17779,41.1936}
Coll Group 26 "ragdoll", interact as 0x0000000000400000 with 0x0000000000000000 has 0
as (22) CONTENTS_SOLID 0xF85771F5
(0) FCOLLISION_FUNC_ENABLE_SOLID_CONTACT
(1) FCOLLISION_FUNC_ENABLE_TRACE_QUERY
(2) FCOLLISION_FUNC_ENABLE_TOUCH_EVENT
Shape Mtl= "flesh" : density 0.0140625, friction 0.8, restitution 0.25
✓ <<< Less
```

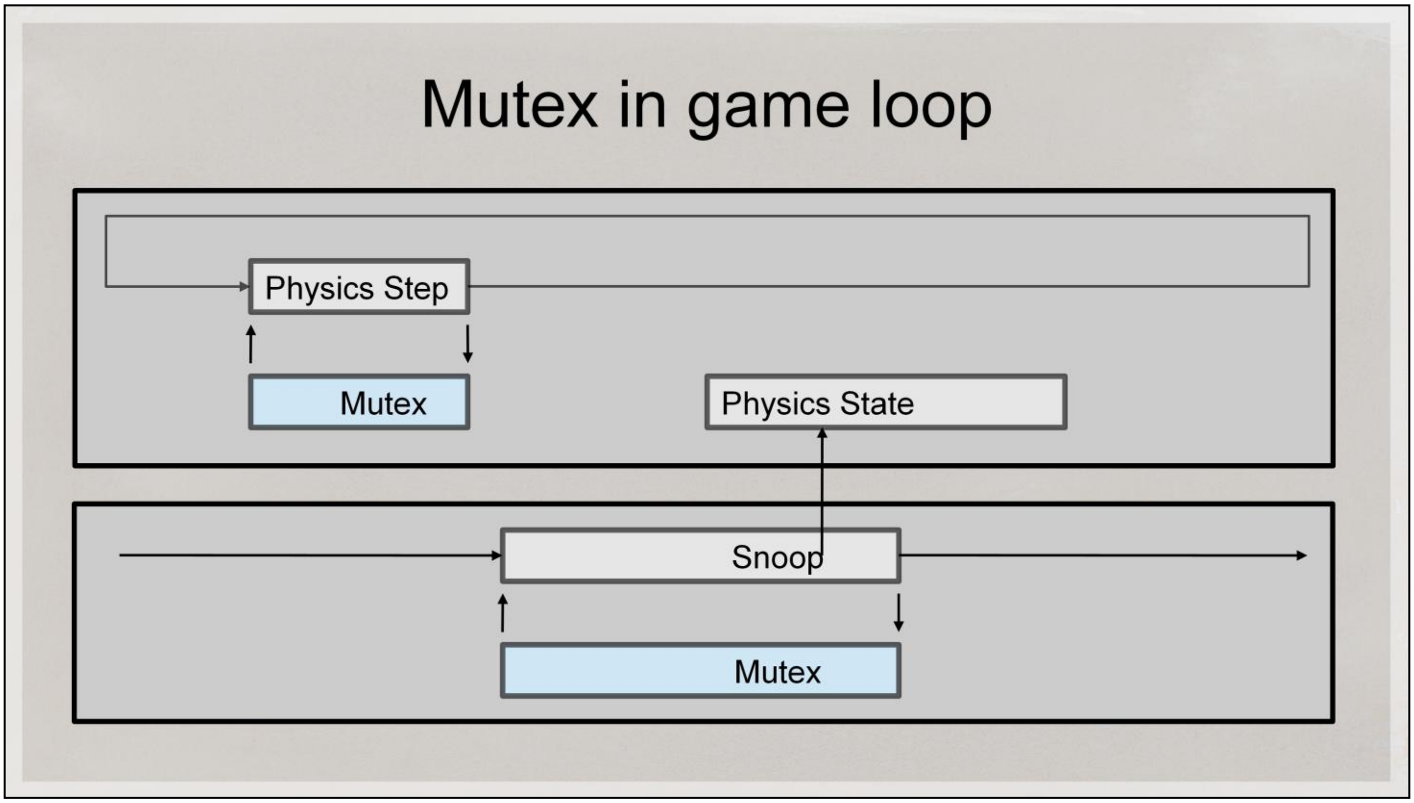
	Body/shape	Min Dist	Points	Lambda	Index
1	Floor	0	0/1	0	6

```
OrthoTwist Joint #1 of 17, handle=0x20002
           #2: Test Ragdoll/ValveBiped.Bip01_Spine2
           #1: Test Ragdoll/ValveBiped.Bip01_Spine1
Twists(XYZ)          -12.4262037   0.4772944   -19.8366833
Twist X Lower:       -10
Twist X Upper:       10
Twist Y Lower:       -10
Twist Y Upper:       10
Twist Z Lower:       -20
Twist Z Upper:       20
✓ Motor Enabled
Motor Max Torque:    519
Motor Velocity:      0.0000000     0.0000000   0.0000000
```

Our Testbed is a tool for divide-and-conquer debugging. It makes it quick and painless.

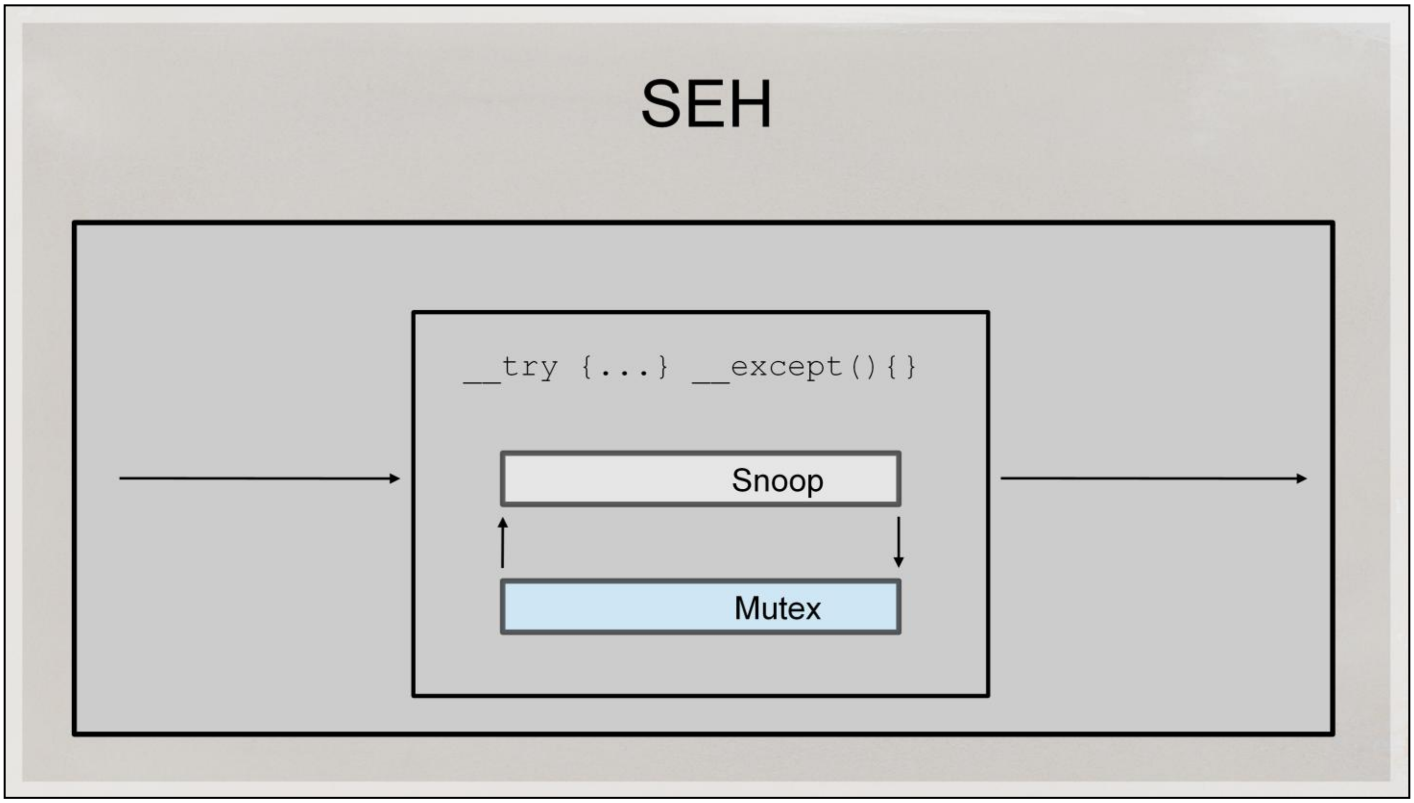
For example, we have a collision filtering system, and time to time something is not colliding right in the game. We just fly around, click on objects that don't collide and drill down to their flags. In most cases we see exactly what's happening within seconds, and you don't have to be a programmer to do that.

Another example: we had an incredibly slow physics. It worked fine, but was very slow. All contacts looked correctly. So we snooped and clicked the objects that were awake. Their contacts looked fine. It took us some seconds to realize there were too many contacts on them.



Mutex in game loop

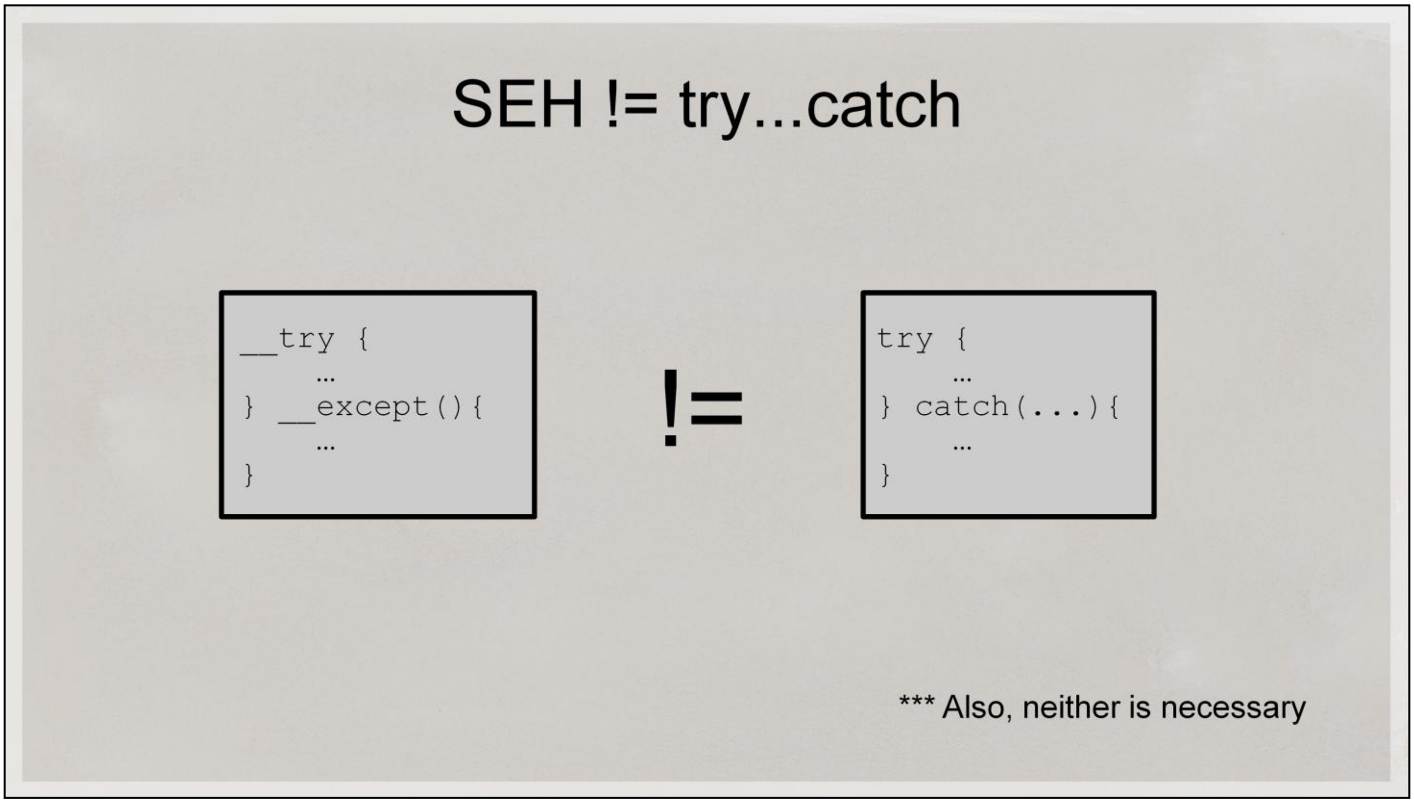
- One consideration here is thread safety
- While you cannot use the Watch window in VS IDE while the game process is running, you can snoop.
- Use Mutex to guard the physics step routine



SEH

```
__try {...} __except(){}
```

- Snooping a running process is tricky: data structures are in flux while you are reading them.
- It's impossible to crash the game by reading data from it. But the snooper might crash if you snoop an inconsistent memory. I'm just using (__try...__except) to catch that. Strictly speaking it's not a safe solution, but it makes the experience smoother.



SEH != try...catch

```
__try {
    …
} __except(){
    …
}
```

!=

```
try {
    …
} catch(...){
    …
}
```

*** Also, neither is necessary

Just to be clear

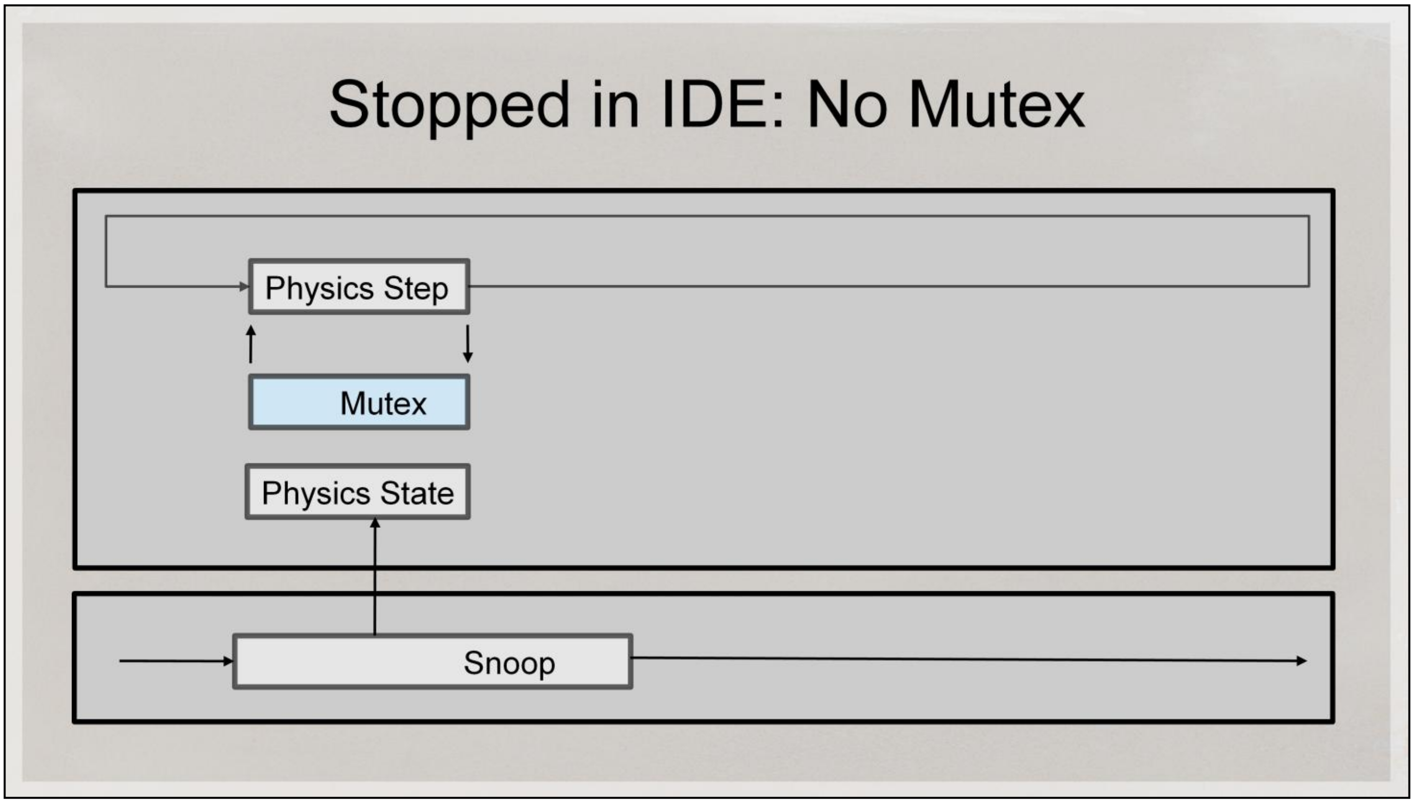
Structured exception handling is not the same as C++ exception handling.

This goes outside the topic of this talk.

In any case, you don't really have to use either of these.

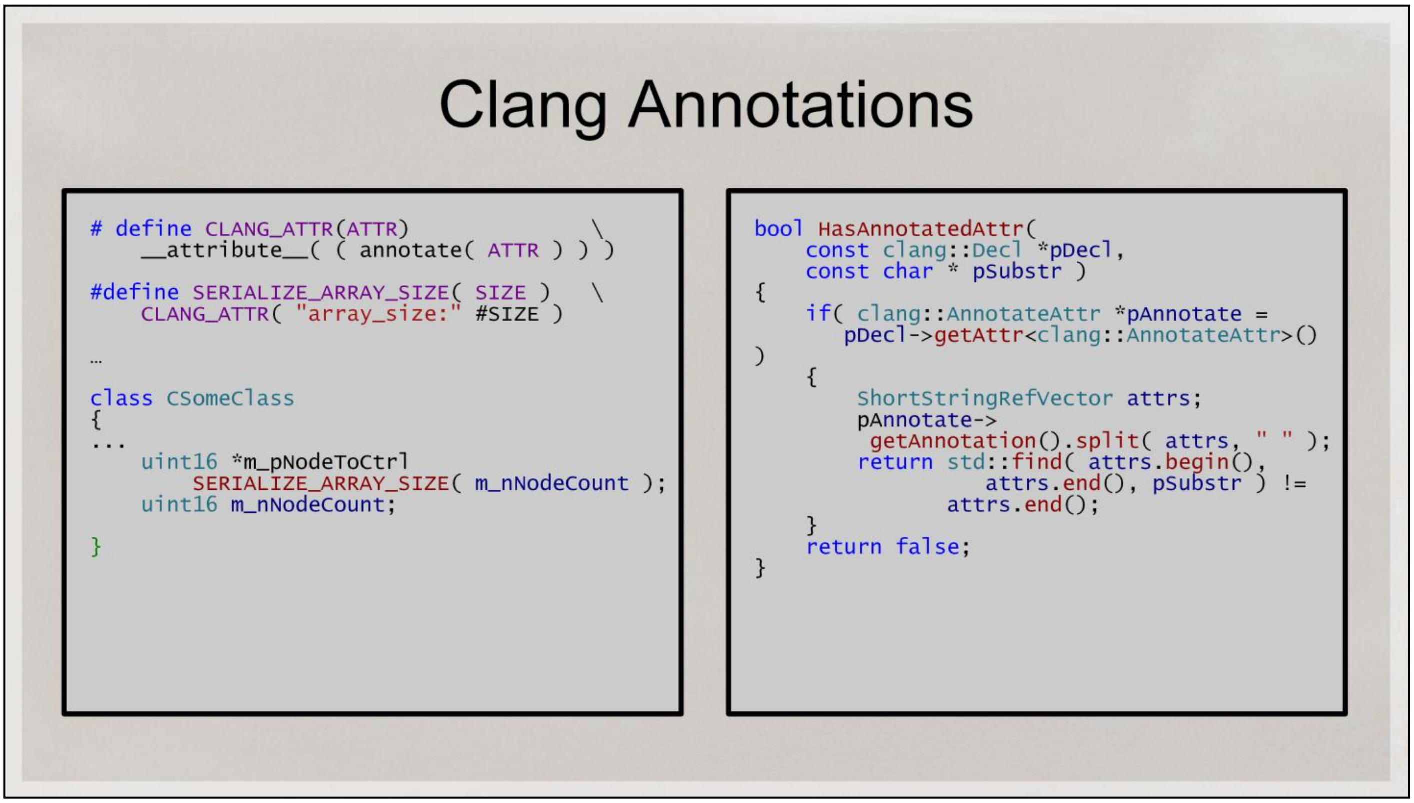
I just find it nice to wrap my sloppy debugging tool code in case I happen to snoop bad data structure.

If I released it as a commercial product, I'd expend the extra effort to detect that memory is unreadable and not crash.



Stopped in IDE: No Mutex

- When the game is stopped on a breakpoint, just ignore the Mutex. Or time it out.
- Data structures cannot change when the game is frozen, so there's a high chance snoop will succeed if they are consistent. It depends on where you put the breakpoint.

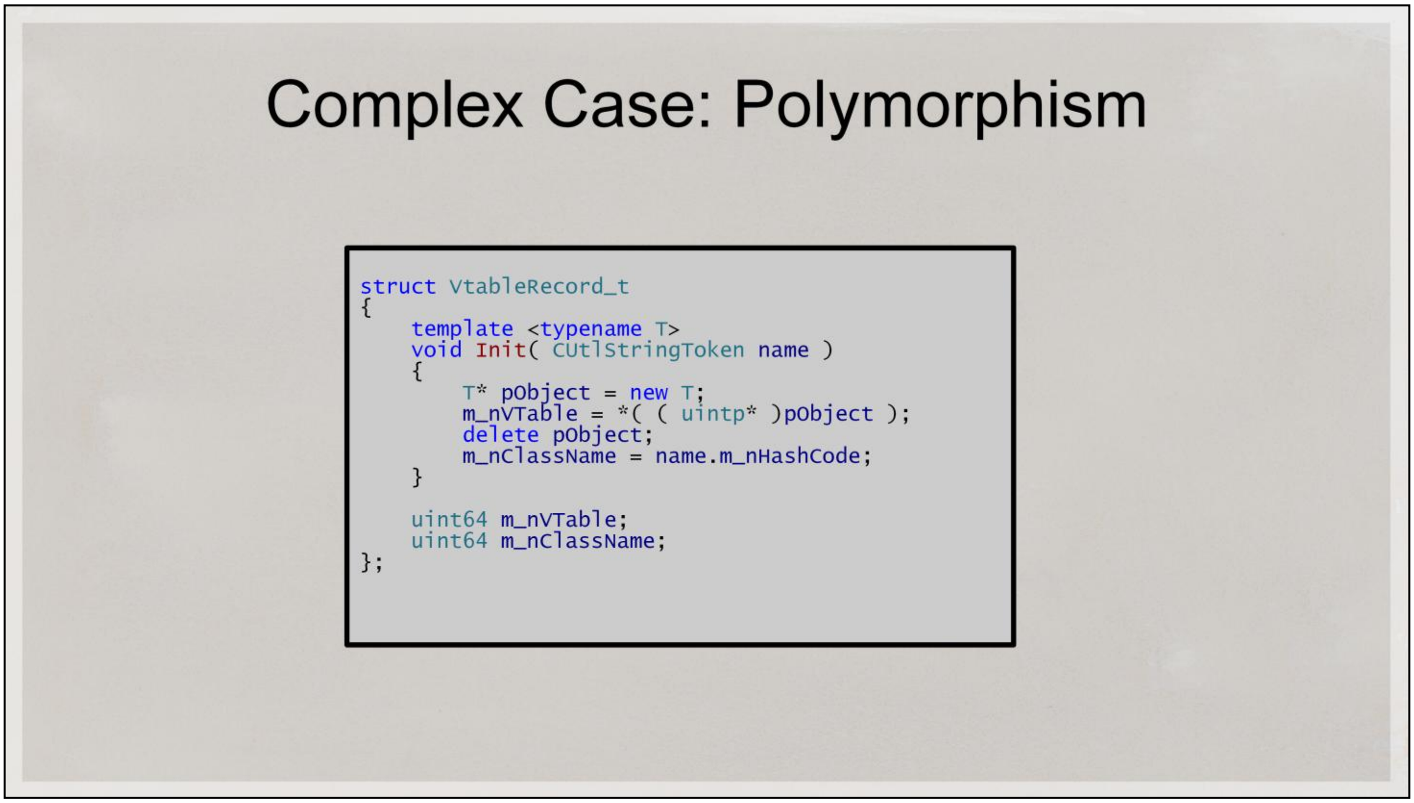


Clang Annotations

```
# define CLANG_ATTR(ATTR)                \
    __attribute__( ( annotate( ATTR ) ) )

#define SERIALIZE_ARRAY_SIZE( SIZE )    \
    CLANG_ATTR( "array_size:" #SIZE )

…

class CSomeClass
{
...
    uint16 *m_pNodeToCtrl
        SERIALIZE_ARRAY_SIZE( m_nNodeCount );
    uint16 m_nNodeCount;

}
```

```
bool HasAnnotatedAttr(
    const clang::Decl *pDecl,
    const char * pSubstr )
{
    if( clang::AnnotateAttr *pAnnotate =
       pDecl->getAttr<clang::AnnotateAttr>()
)
    {
        ShortStringRefVector attrs;
        pAnnotate->
         getAnnotation().split( attrs, " " );
        return std::find( attrs.begin(),
                    attrs.end(), pSubstr ) !=
                 attrs.end();
    }
    return false;
}
```

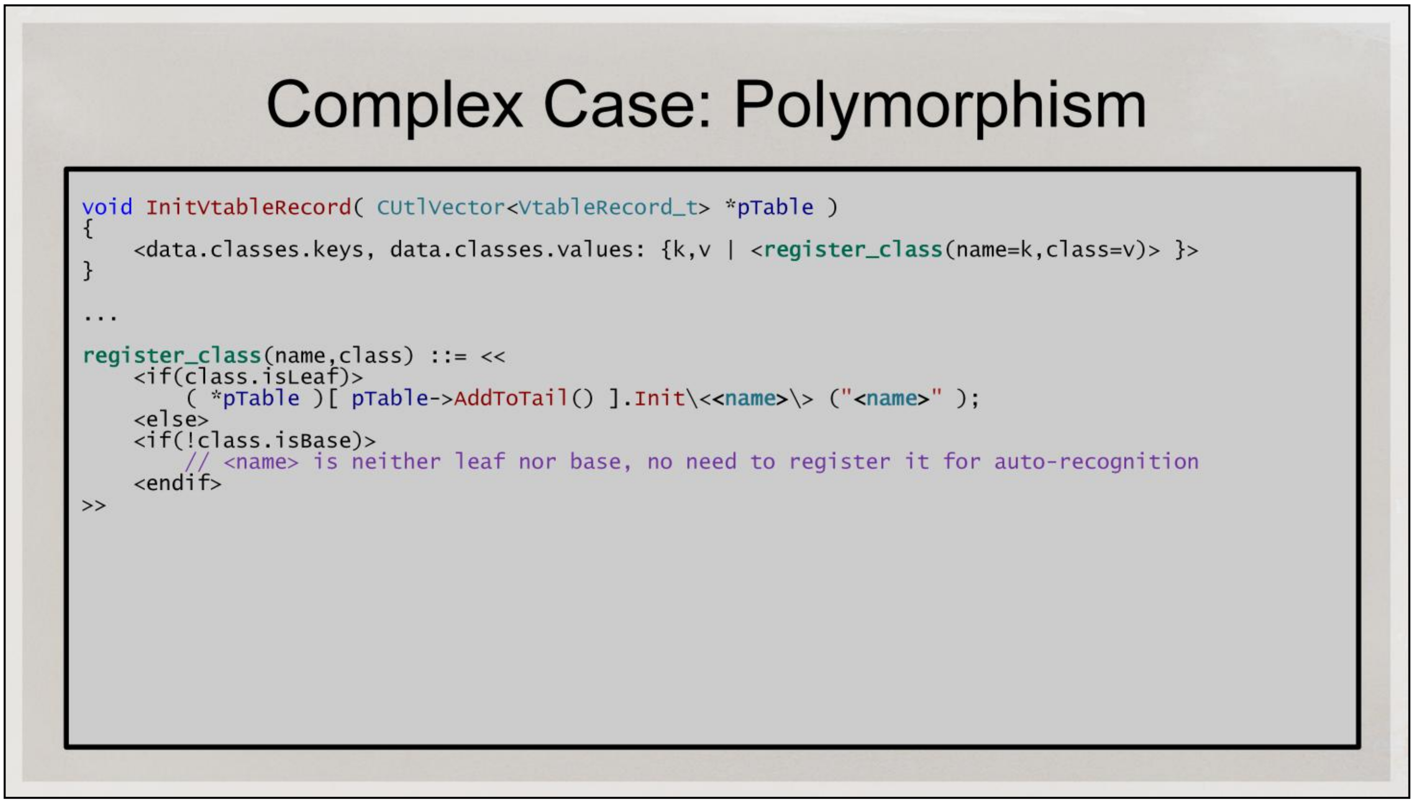
- Sometimes we need some annotations to our code. E.g. for each pointer you serializer will need to know how many objects it's pointing to. It's easy to put this in (see the left slide).
- And ot's easy to add and read those annotations in the parser (see the right slide)



Complex Case: Polymorphism

```
struct VtableRecord_t
{
    template <typename T>
    void Init( CUtlStringToken name )
    {
        T* pObject = new T;
        m_nVTable = *( ( uintp* )pObject );
        delete pObject;
        m_nClassName = name.m_nHashCode;
    }

    uint64 m_nVTable;
    uint64 m_nClassName;
};
```

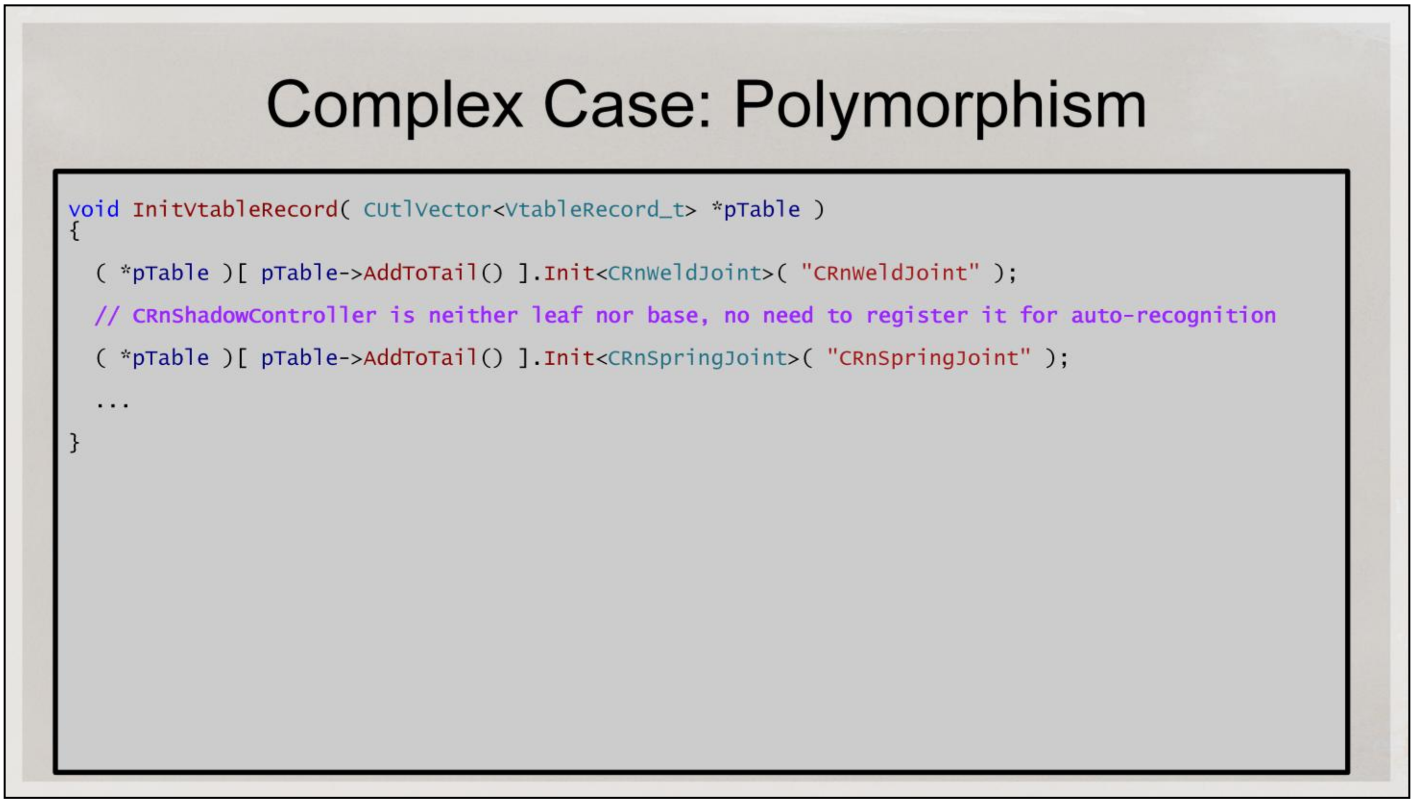
- I'd like to talk about a couple of less trivial cases, like classes with Vtables.
- I auto-generate a function that creates an instance of each class with Vtable, and copies its Vtable point into a known place (an array of uintptr_t).
- The snooper can then read the array and use it to recognize class type by its vtable pointer. It's like RTTI that works across process boundaries.



Complex Case: Polymorphism

```
void InitVtableRecord( CUtlVector<VtableRecord_t> *pTable )
{
    <data.classes.keys, data.classes.values: {k,v | <register_class(name=k,class=v)> }>
}

...

register_class(name,class) ::= <<
    <if(class.isLeaf)>
        ( *pTable )[ pTable->AddToTail() ].Init\<<name>\> ("<name>" );
    <else>
    <if(!class.isBase)>
        // <name> is neither leaf nor base, no need to register it for auto-recognition
    <endif>
>>
```

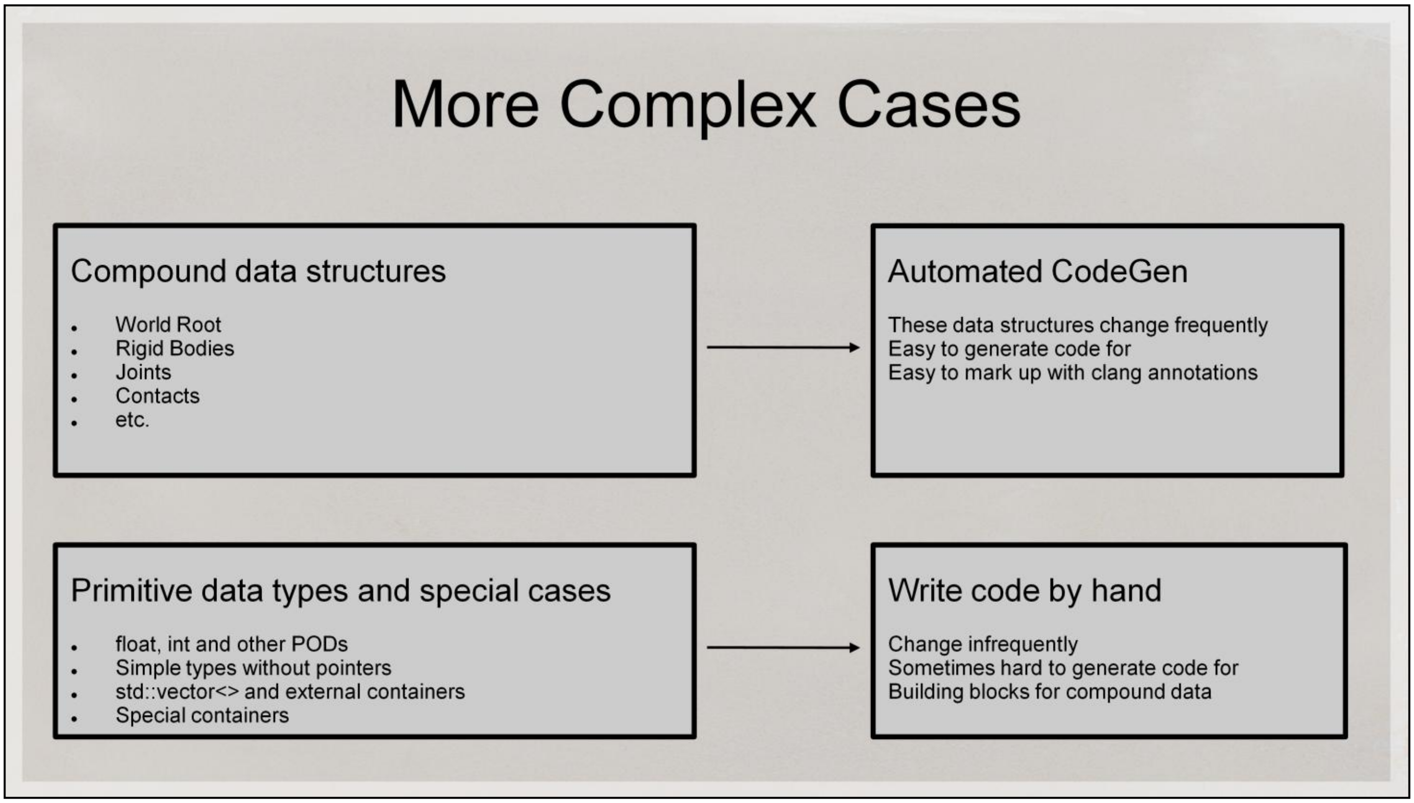
- This is how I generate that code with StringTemplates



Complex Case: Polymorphism

```
void InitVtableRecord( CUtlVector<VtableRecord_t> *pTable )
{
  ( *pTable )[ pTable->AddToTail() ].Init<CRnWeldJoint>( "CRnWeldJoint" );
  // CRnShadowController is neither leaf nor base, no need to register it for auto-recognition
  ( *pTable )[ pTable->AddToTail() ].Init<CRnSpringJoint>( "CRnSpringJoint" );
  ...
}
```

- And this is what the generated code looks like. One line is generated for every class with vtable. It’s done automatically, so you don’t have to remember to add a class to the list once you add a virtual .
- You don’t have to remove a class when you remove the last virtual from it either.



More Complex Cases

Compound data structures

- World Root
- Rigid Bodies
- Joints
- Contacts
- etc.

Automated CodeGen

These data structures change frequently
Easy to generate code for
Easy to mark up with clang annotations

Primitive data types and special cases

- float, int and other PODs
- Simple types without pointers
- std::vector<> and external containers
- Special containers

Write code by hand

Change infrequently
Sometimes hard to generate code for
Building blocks for compound data

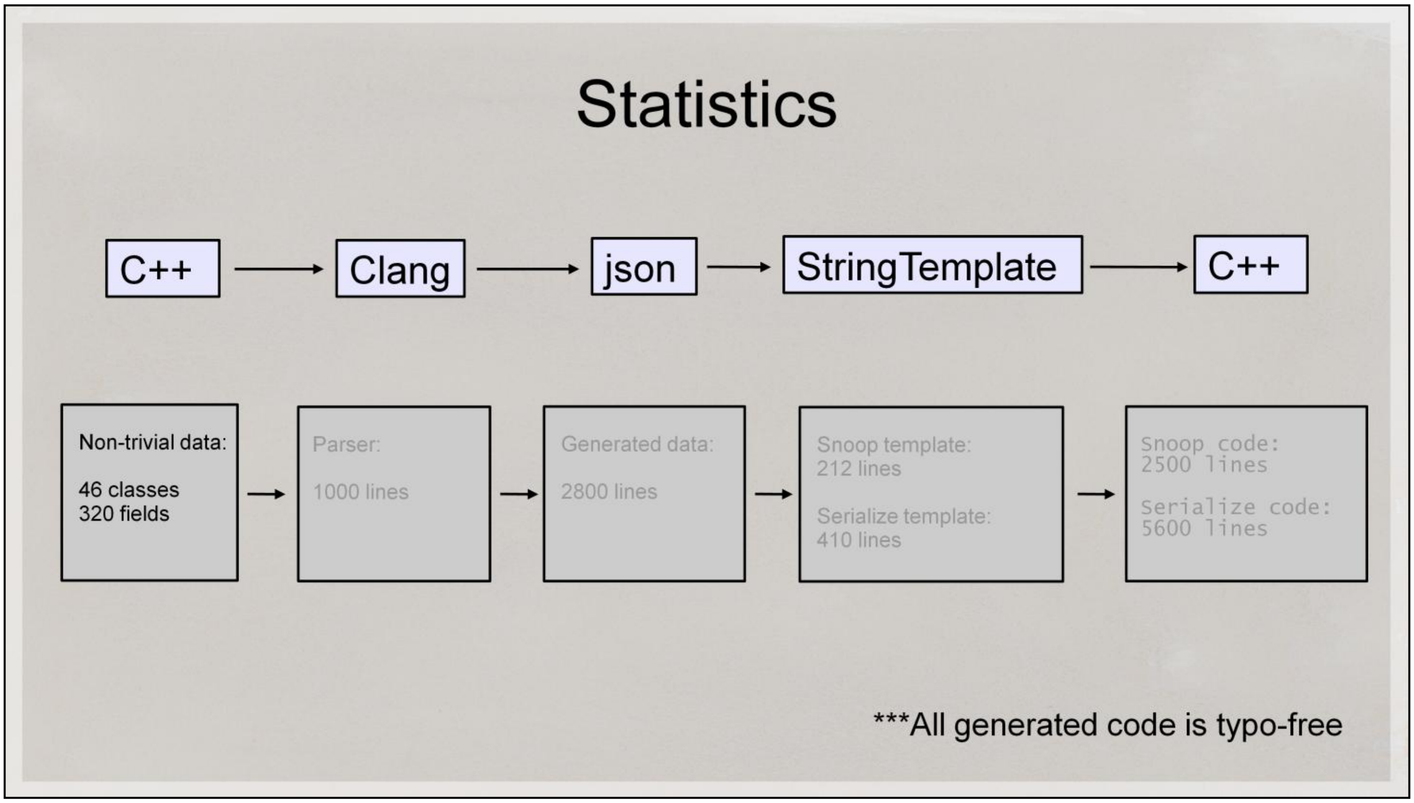
For some data structures, it's just easier to write serialization code by hand. The code generator should know about leafy data types like int and float, and generate serialization code automatically. There's no problem with structures only consisting of those data types, those can be simply copied directly from another process address space.

It's a bit more involved with pointers. In general, C++ does not provide enough semantics for you to know if a pointer points to one object, or a vector of objects. I'm using clang annotation to add that semantics. So, the code generator looks at that annotation and computes the size of the array and serializes accordingly.

We don't use multiple heaps for our physics engine, but if you do, you'll have to provide semantics about how to allocate the snooped or unserialized data structures.

In strange cases, like an int that is a pointer, but the lower 2 bits have a special meaning, you probably want to write that routine by hand. Also when you have a union and have to run code to decide what it means.

You have a choice whether to make your code auto-generator smarter (and harder to maintain) or snoop a specific class manually. It's a judgement call.



Statistics

C++ → Clang → json → StringTemplate → C++

Non-trivial data:

46 classes
320 fields

Parser:

1000 lines

Generated data:

2800 lines

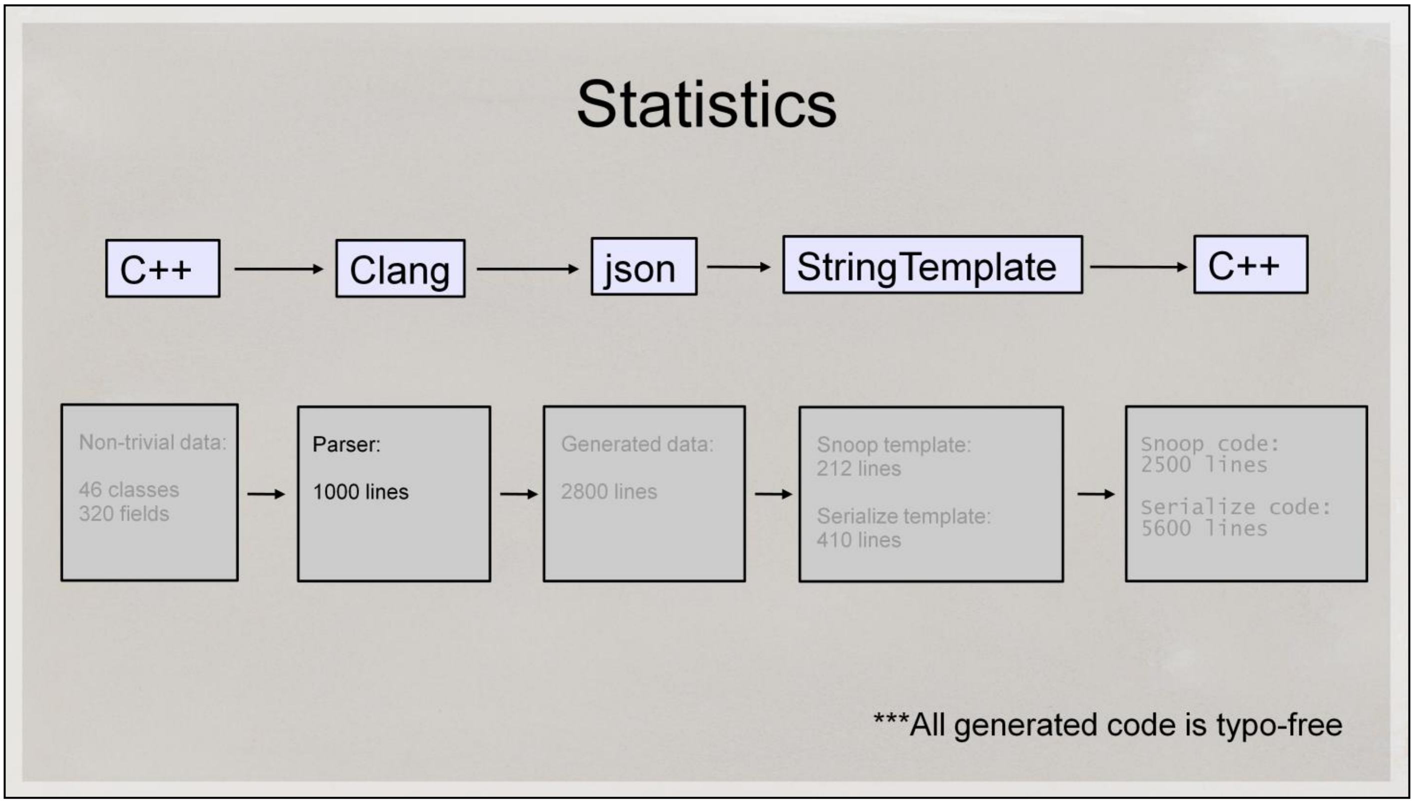
Snoop template:
212 lines

Serialize template:
410 lines

```
Snoop code:
2500 lines

Serialize code:
5600 lines
```

***All generated code is typo-free

- We currently have 46 non-trivial classes with 320 fields for which serializer and snooper code is generated.



Statistics

C++ → Clang → json → StringTemplate → C++

Non-trivial data:

46 classes
320 fields

Parser:

1000 lines

Generated data:

2800 lines

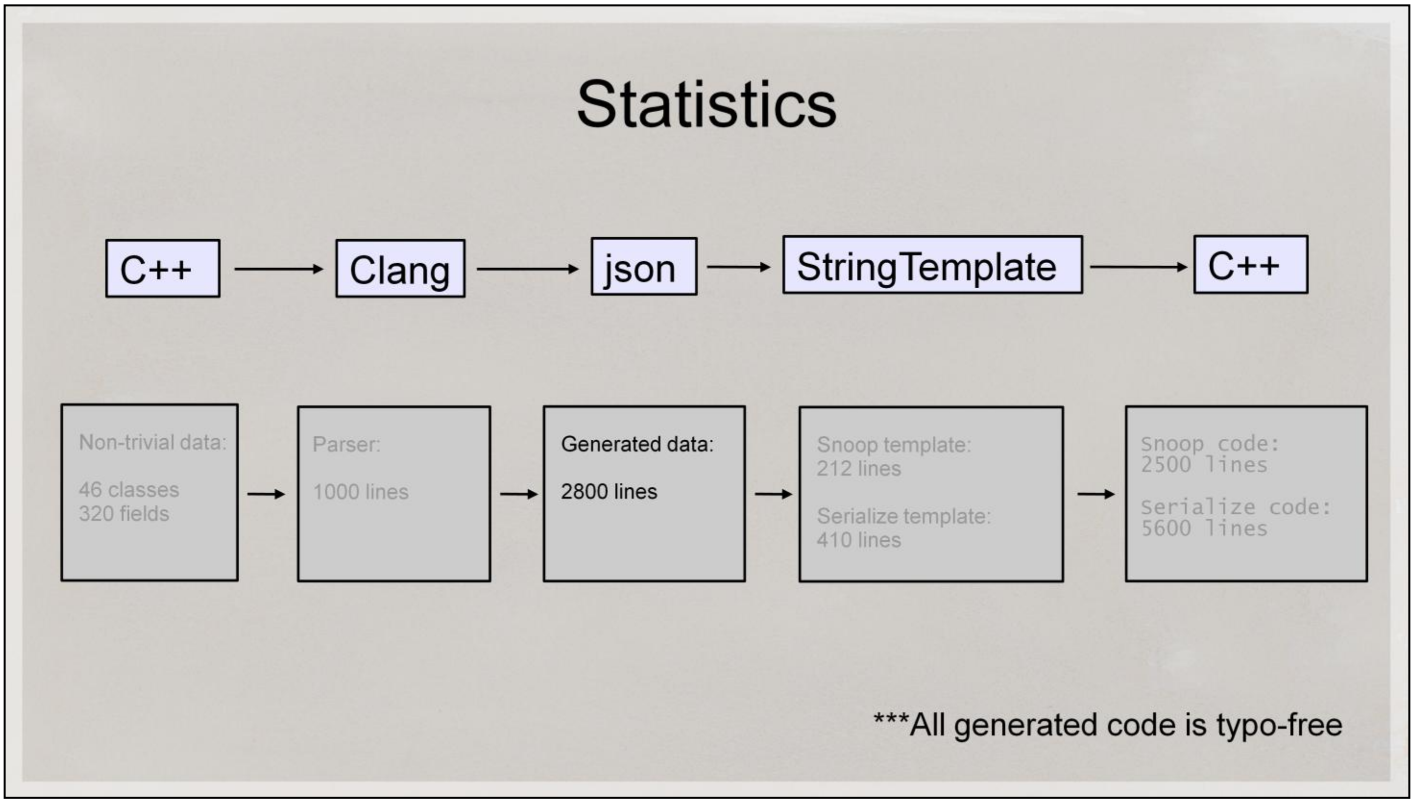
Snoop template:
212 lines

Serialize template:
410 lines

```
Snoop code:
2500 lines

Serialize code:
5600 lines
```

***All generated code is typo-free

- The parser that uses clang is about 1000 lines long. I probably spent a couple of weeks part time writing it, and I've been using it for a year.



Statistics

C++ → Clang → json → StringTemplate → C++

Non-trivial data:

46 classes
320 fields

Parser:

1000 lines

Generated data:

2800 lines

Snoop template:
212 lines

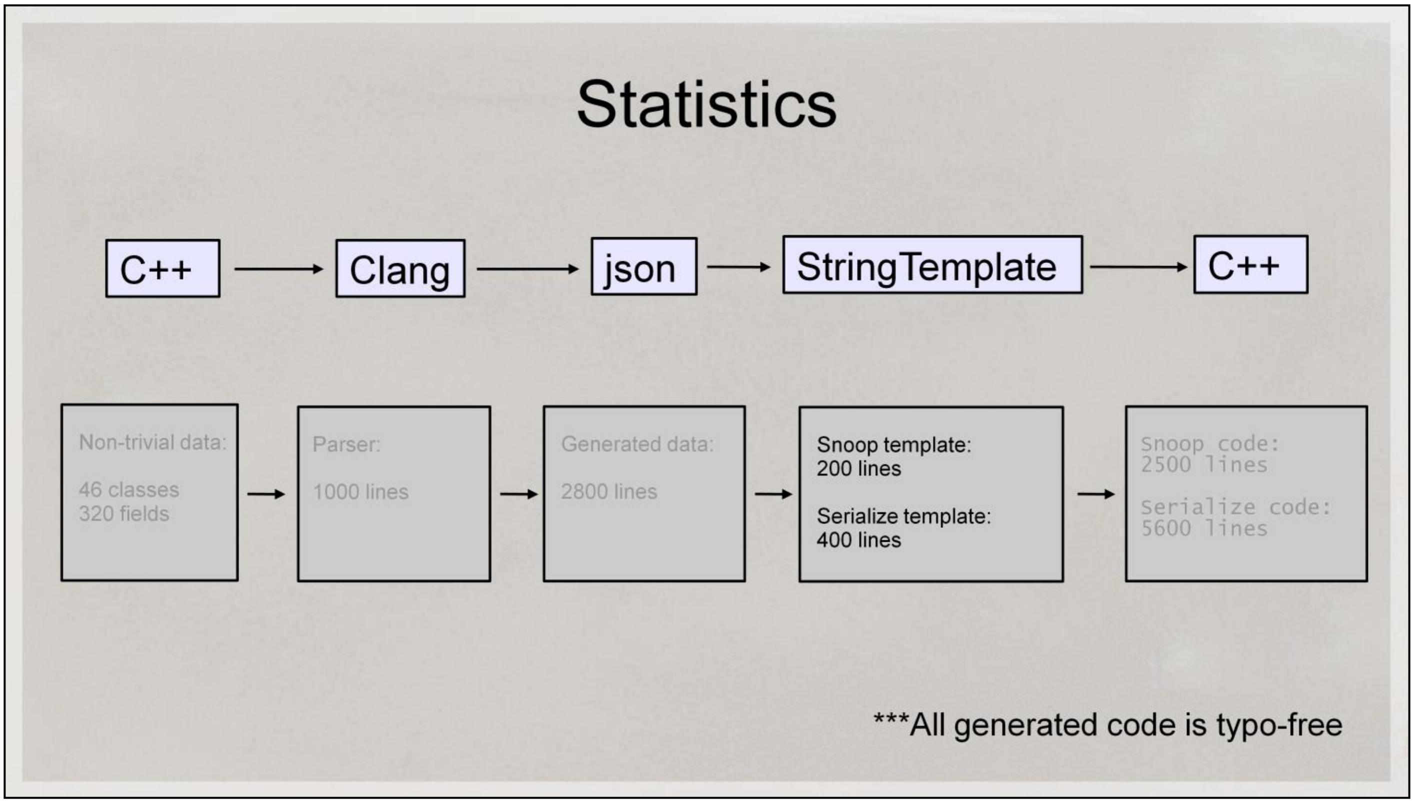
Serialize template:
410 lines

Snoop code:
2500 lines

Serialize code:
5600 lines

***All generated code is typo-free

- The generated json is 2800 lines long



Statistics

C++ → Clang → json → StringTemplate → C++

Non-trivial data:

46 classes
320 fields

Parser:

1000 lines

Generated data:

2800 lines

Snoop template:
200 lines

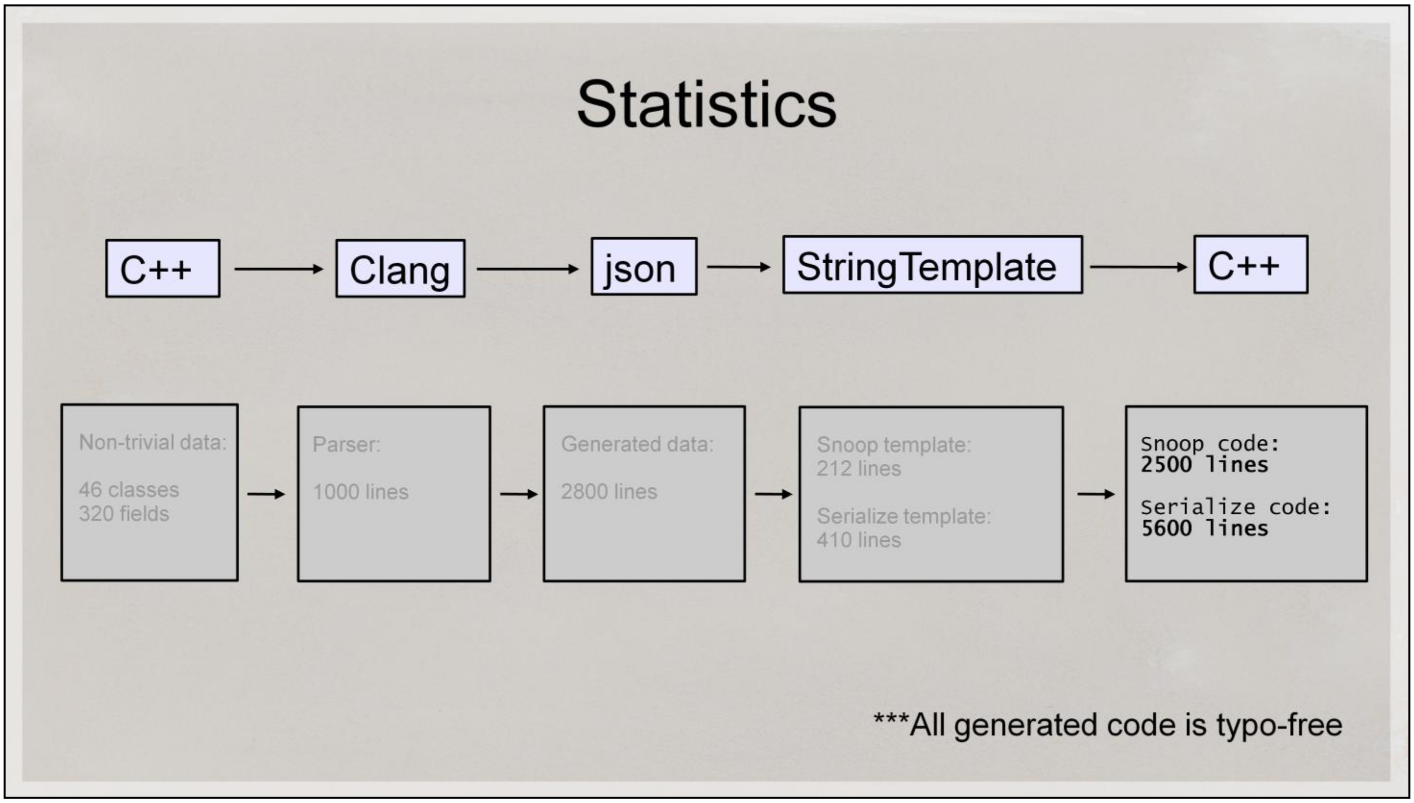
Serialize template:
400 lines

Snoop code:
2500 lines

Serialize code:
5600 lines

***All generated code is typo-free

- Snooper template is 200 lineslong .
- Serializer template is 400 lines.



Statistics

C++ → Clang → json → StringTemplate → C++

Non-trivial data:

46 classes
320 fields

→

Parser:

1000 lines

→

Generated data:

2800 lines

→

Snoop template:
212 lines

Serialize template:
410 lines

→

Snoop code:
2500 lines

Serialize code:
5600 lines

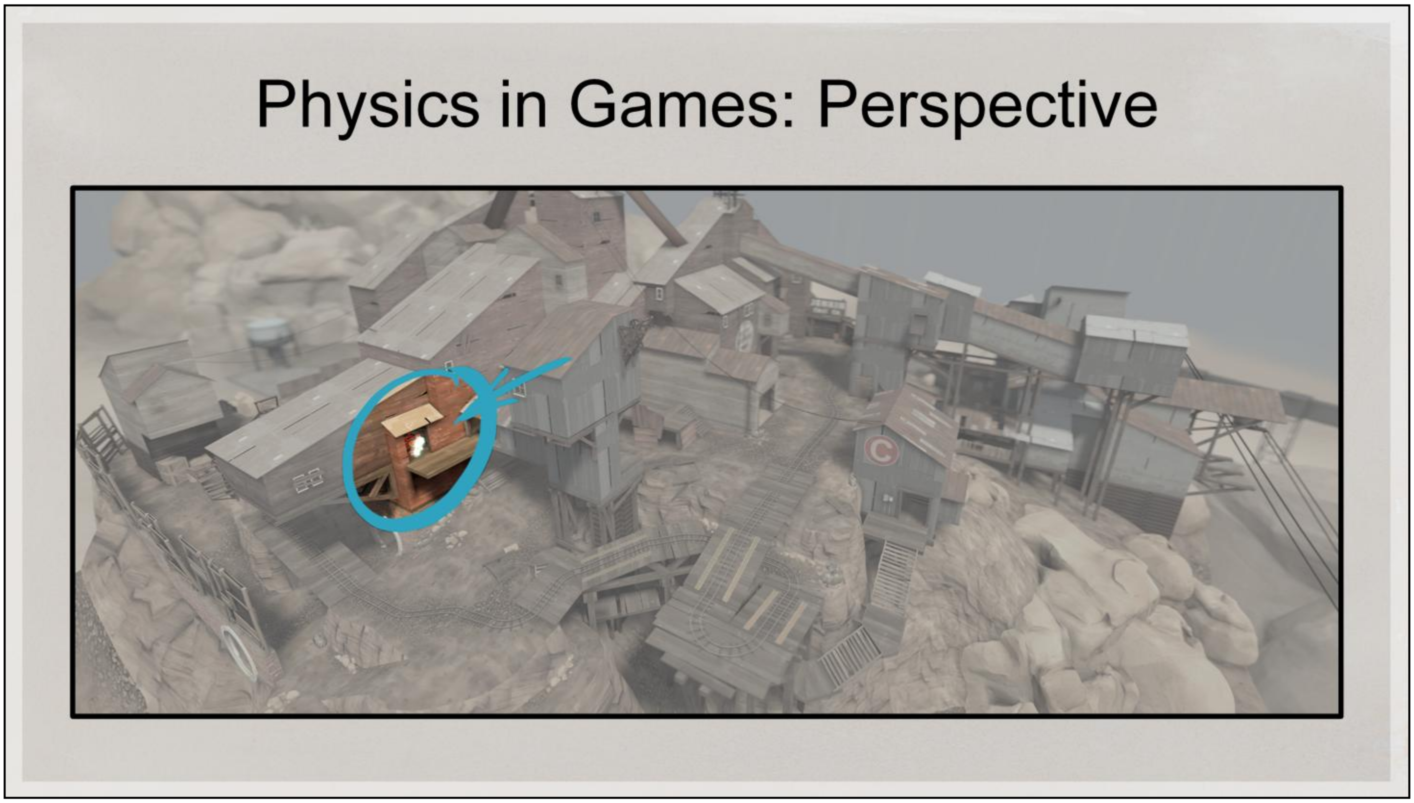
***All generated code is typo-free

- Auto-generated snooper is 2500 lines long.
- Serializer is 5600 lines long, because it includes 3 directions: serialize to stream, unserialize from stream, count number of bytes used (very similar to Serialize, but actually counts allocated memory).
- It takes 2-8ms to snoop a typical game frame, which is normally maybe 10 mb large

Physics in Games: Integration

Every game physics engine eventually gets integrated into a game. It makes change-compile-test iteration much longer. Especially when artists start building a huge map, and new exciting bugs show up that only happen in this huge map once in a blue moon.

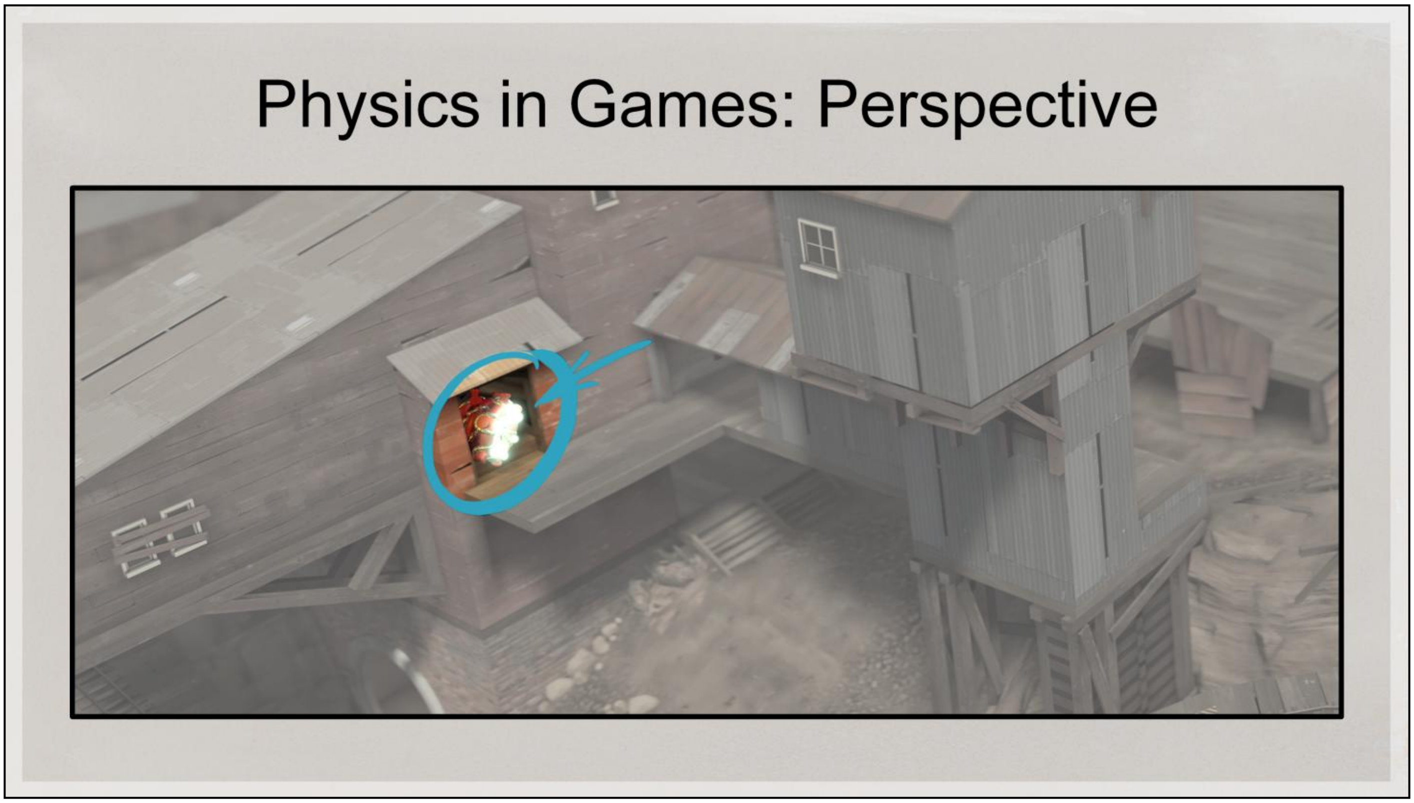
It makes bug hunting mind-numbingly slow. Because the game code is much, much larger than the physics code we know so well.



Physics in Games: Perspective

Game code is written by many people over many years. Physics engine is self-contained and neat, almost miniature in comparison.

At valve, the sheer amount of logic in the old game code with history is very large.

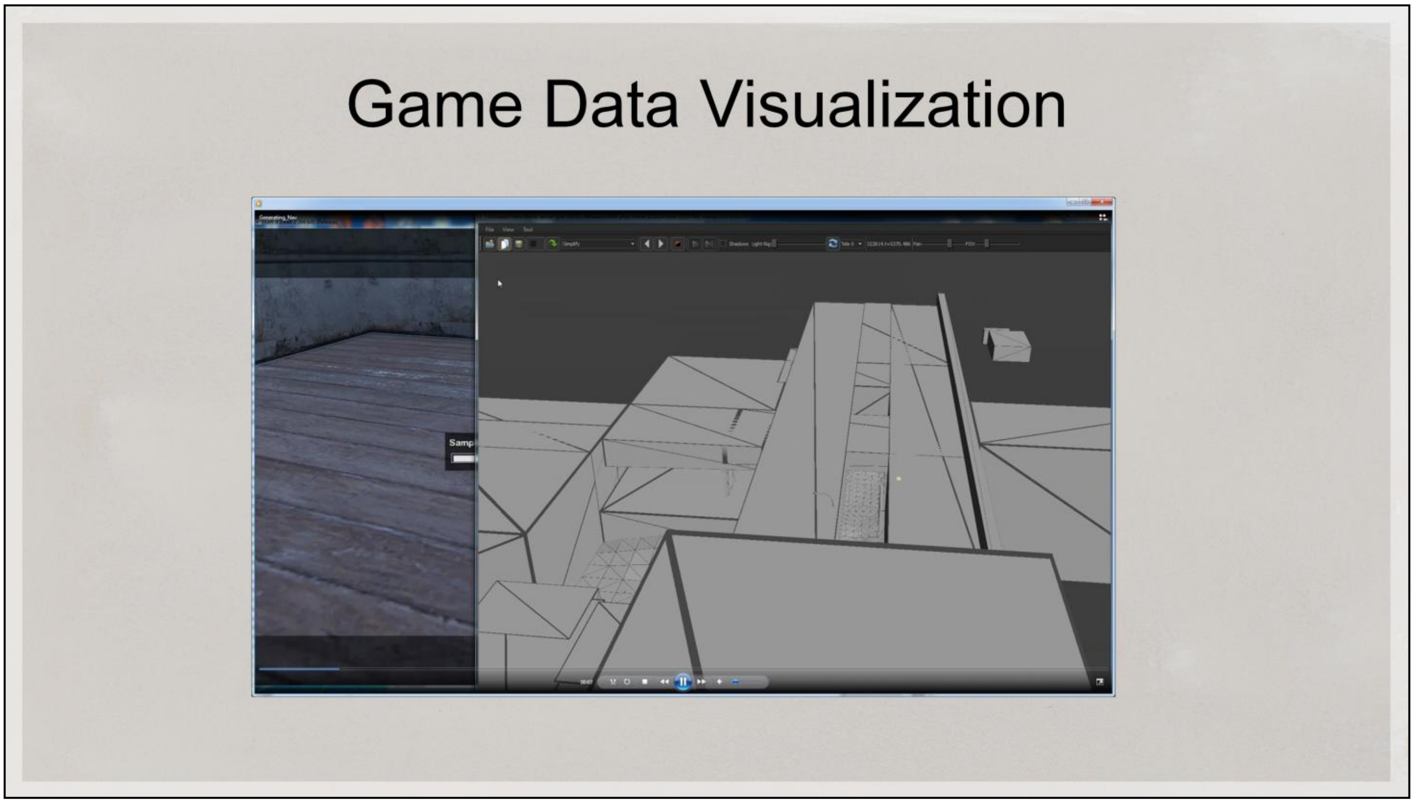


Physics in Games: Perspective

It's really easy to lose perspective when you work on physics for a long time.

The ultimate goal of physics in a game is to make the game better. It's not to make the best or fastest or true to life physics simulation. That is why in-game debugging and optimization is very important.

If the game wants to cast a lot of rays, we need to optimize that first. Games don't generally have stacks of cubes everywhere, so that's not the best benchmark for a game physics engine. It's best to put it in the game and debug and profile it in the game, if you want it to perform well in the game.



Game Data Visualization

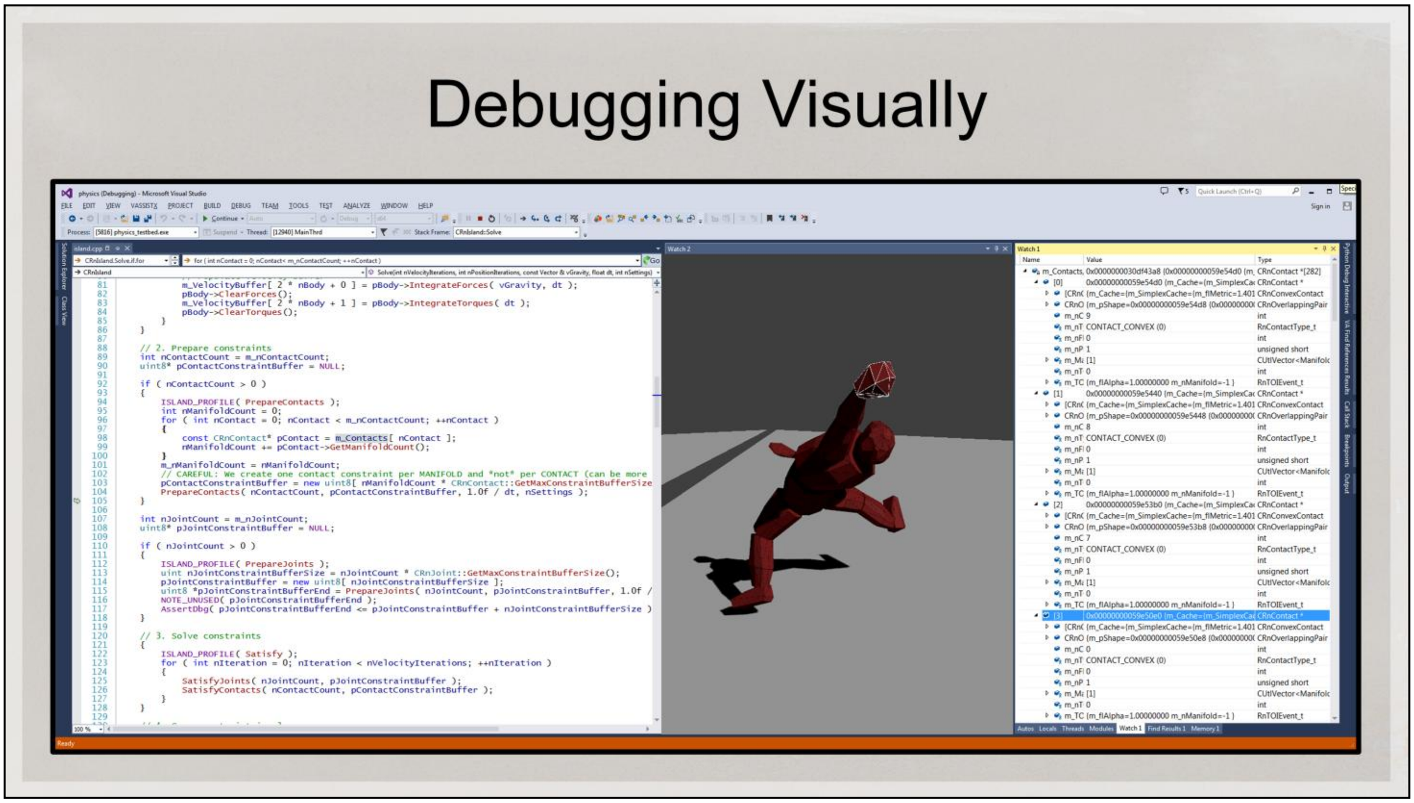
Run Generating_Nav

Here's another example of an unexpected benefit of visualizing everything in physics engine. Testbed records and displays all traces. And left4dead navigation system does a lot of traces. So, when connected to the game, Testbed visualizes an AI algorithm.

Set Your Priorities

I suggest we can all spend a couple of weeks up front to make our life more comfortable and deliver a better product. Sometimes you can't do that, but generally if your time horizon is years, you can afford a couple of weeks.

What I'm describing is pretty general technique. Game physics is just an example, but it's usable in any complex software.



Debugging Visually

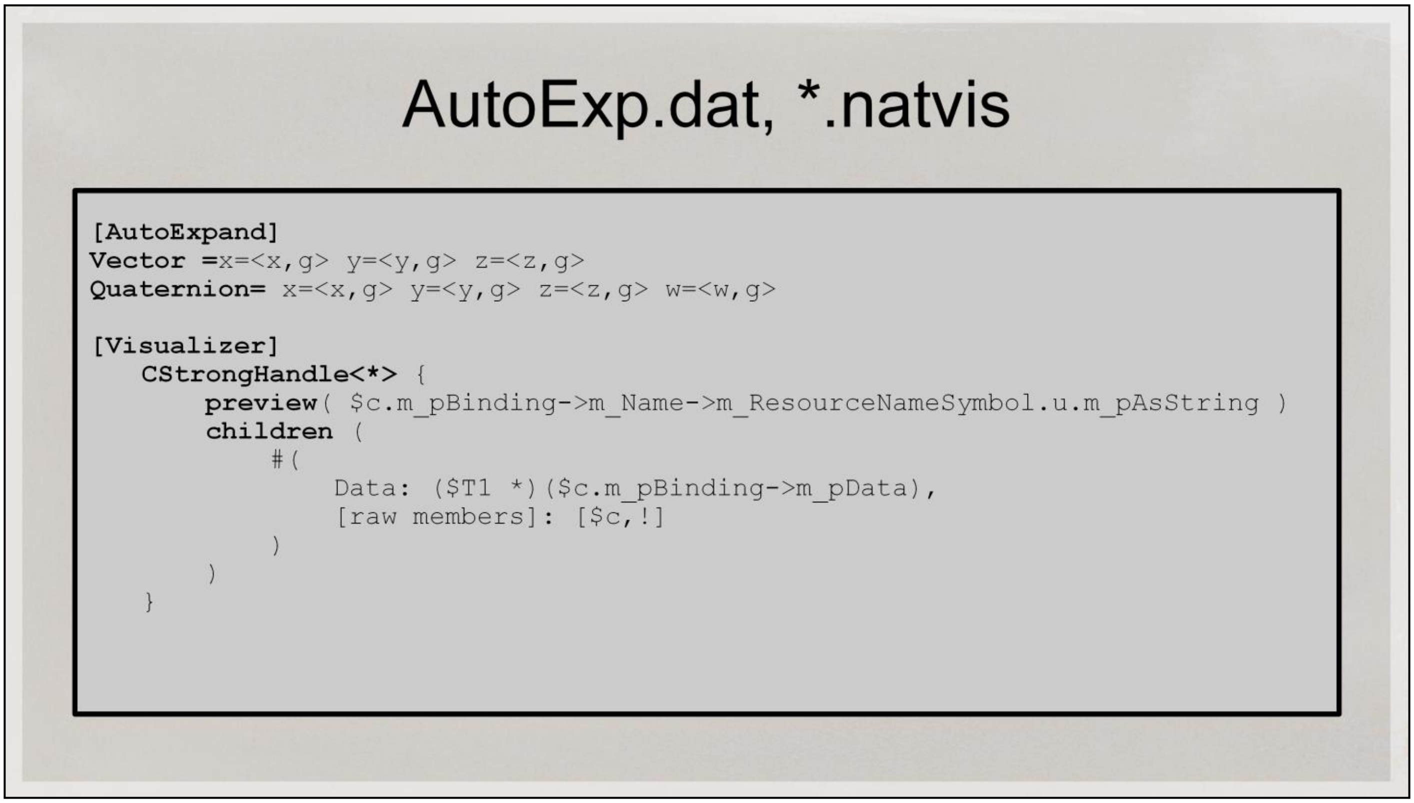
physics (Debugging) - Microsoft Visual Studio
FILE EDIT VIEW VASSISTX PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
Continue
Quick Launch (Ctrl+Q)
Sign in
Process: [5816] physics_testbed.exe
Suspend
Thread: [12940] MainThrd
Stack Frame: CRnIsland::Solve
island.cpp
CRnIsland.Solve.if.for
for (int nContact = 0; nContact< m_nContactCount; ++nContact)
Go
CRnIsland
Solve(int nVelocityIterations, int nPositionIterations, const Vector & vGravity, float dt, int nSettings)
Solution Explorer
Class View
Watch 2
Watch 1
Python Debug Interactive
VA Find References Results
Call Stack
Breakpoints
Output
100 %
Ready
Autos
Locals
Threads
Modules
Watch 1
Find Results 1
Memory 1

```
            m_VelocityBuffer[ 2 * nBody + 0 ] = pBody->IntegrateForces( vGravity, dt );
            pBody->ClearForces();
            m_VelocityBuffer[ 2 * nBody + 1 ] = pBody->IntegrateTorques( dt );
            pBody->ClearTorques();
        }
    }

    // 2. Prepare constraints
    int nContactCount = m_nContactCount;
    uint8* pContactConstraintBuffer = NULL;

    if ( nContactCount > 0 )
    {
        ISLAND_PROFILE( PrepareContacts );
        int nManifoldCount = 0;
        for ( int nContact = 0; nContact < m_nContactCount; ++nContact )
        {
            const CRnContact* pContact = m_Contacts[ nContact ];
            nManifoldCount += pContact->GetManifoldCount();
        }
        m_nManifoldCount = nManifoldCount;
        // CAREFUL: We create one contact constraint per MANIFOLD and *not* per CONTACT (can be more
        pContactConstraintBuffer = new uint8[ nManifoldCount * CRnContact::GetMaxConstraintBufferSize
        PrepareContacts( nContactCount, pContactConstraintBuffer, 1.0f / dt, nSettings );
    }

    int nJointCount = m_nJointCount;
    uint8* pJointConstraintBuffer = NULL;

    if ( nJointCount > 0 )
    {
        ISLAND_PROFILE( PrepareJoints );
        uint nJointConstraintBufferSize = nJointCount * CRnJoint::GetMaxConstraintBufferSize();
        pJointConstraintBuffer = new uint8[ nJointConstraintBufferSize ];
        uint8 *pJointConstraintBufferEnd = PrepareJoints( nJointCount, pJointConstraintBuffer, 1.0f /
        NOTE_UNUSED( pJointConstraintBufferEnd );
        AssertDbg( pJointConstraintBufferEnd <= pJointConstraintBuffer + nJointConstraintBufferSize )
    }

    // 3. Solve constraints
    {
        ISLAND_PROFILE( Satisfy );
        for ( int nIteration = 0; nIteration < nVelocityIterations; ++nIteration )
        {
            SatisfyJoints( nJointCount, pJointConstraintBuffer );
            SatisfyContacts( nContactCount, pContactConstraintBuffer );
        }
    }
```

Name	Value	Type
m_Contacts,	0x0000000030df43a8 {0x00000000059e54d0 {m_	CRnContact *[282]
[0]	0x00000000059e54d0 {m_Cache={m_SimplexCa	CRnContact *
[CRn(	{m_Cache={m_SimplexCache={m_flMetric=1.401	CRnConvexContact
CRnO	{m_pShape=0x00000000059e54d8 {0x00000000(	CRnOverlappingPair
m_nC	9	int
m_nT	CONTACT_CONVEX (0)	RnContactType_t
m_nFl	0	int
m_nP	1	unsigned short
m_Ma	[1]	CUtlVector<Manifolc
m_nT	0	int
m_TC	{m_flAlpha=1.00000000 m_nManifold=-1 }	RnTOIEvent_t
[1]	0x00000000059e5440 {m_Cache={m_SimplexCa	CRnContact *
[CRn(	{m_Cache={m_SimplexCache={m_flMetric=1.401	CRnConvexContact
CRnO	{m_pShape=0x00000000059e5448 {0x00000000(	CRnOverlappingPair
m_nC	8	int
m_nT	CONTACT_CONVEX (0)	RnContactType_t
m_nFl	0	int
m_nP	1	unsigned short
m_Ma	[1]	CUtlVector<Manifolc
m_nT	0	int
m_TC	{m_flAlpha=1.00000000 m_nManifold=-1 }	RnTOIEvent_t
[2]	0x00000000059e53b0 {m_Cache={m_SimplexCa	CRnContact *
[CRn(	{m_Cache={m_SimplexCache={m_flMetric=1.401	CRnConvexContact
CRnO	{m_pShape=0x00000000059e53b8 {0x00000000(	CRnOverlappingPair
m_nC	7	int
m_nT	CONTACT_CONVEX (0)	RnContactType_t
m_nFl	0	int
m_nP	1	unsigned short
m_Ma	[1]	CUtlVector<Manifolc
m_nT	0	int
m_TC	{m_flAlpha=1.00000000 m_nManifold=-1 }	RnTOIEvent_t
[3]	0x00000000059e50e0 {m_Cache={m_SimplexCa	CRnContact *
[CRn(	{m_Cache={m_SimplexCache={m_flMetric=1.401	CRnConvexContact
CRnO	{m_pShape=0x00000000059e50e8 {0x00000000(	CRnOverlappingPair
m_nC	0	int
m_nT	CONTACT_CONVEX (0)	RnContactType_t
m_nFl	0	int
m_nP	1	unsigned short
m_Ma	[1]	CUtlVector<Manifolc
m_nT	0	int
m_TC	{m_flAlpha=1.00000000 m_nManifold=-1 }	RnTOIEvent_t

Eventually, I want my Visual Studio to look like this when I'm debugging.

You can make a debugger extension with VS Extensibility, and I might make one some day. But when I experimented with it, it was very inconvenient to write a VS extension of such complexity.

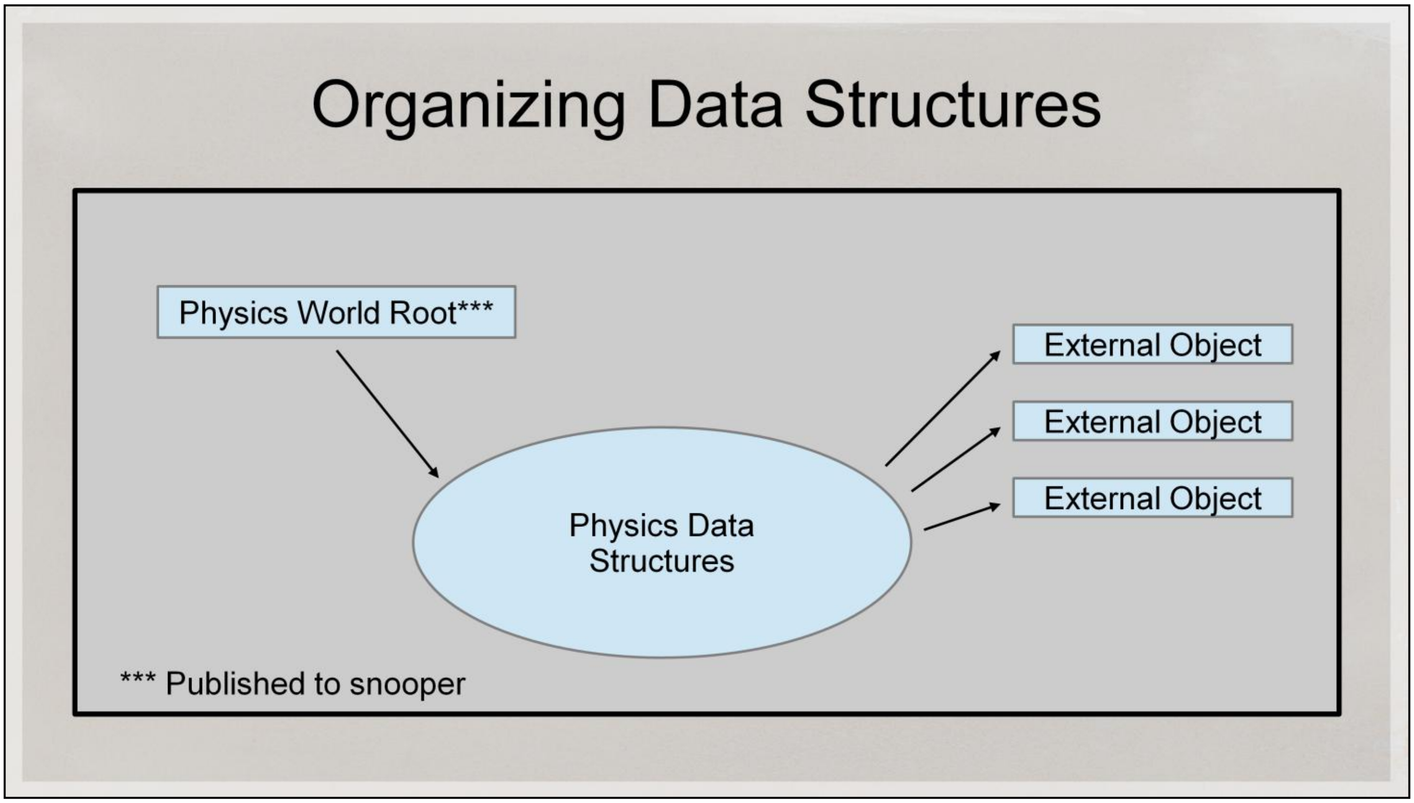


AutoExp.dat, *.natvis

```
[AutoExpand]
Vector =x=<x,g> y=<y,g> z=<z,g>
Quaternion= x=<x,g> y=<y,g> z=<z,g> w=<w,g>

[Visualizer]
   CStrongHandle<*> {
        preview( $c.m_pBinding->m_Name->m_ResourceNameSymbol.u.m_pAsString )
        children (
            #(
                Data: ($T1 *)($c.m_pBinding->m_pData),
                [raw members]: [$c,!]
            )
        )
   }
```

I hope everyone knows about autoexp.dat. It lets you visualize in text, so to say.

I strongly suggest you just put all your data structures in there. Try to make them more readable - in your Watch window. It helps a lot.



Organizing Data Structures

Physics World Root***

Physics Data Structures

External Object

External Object

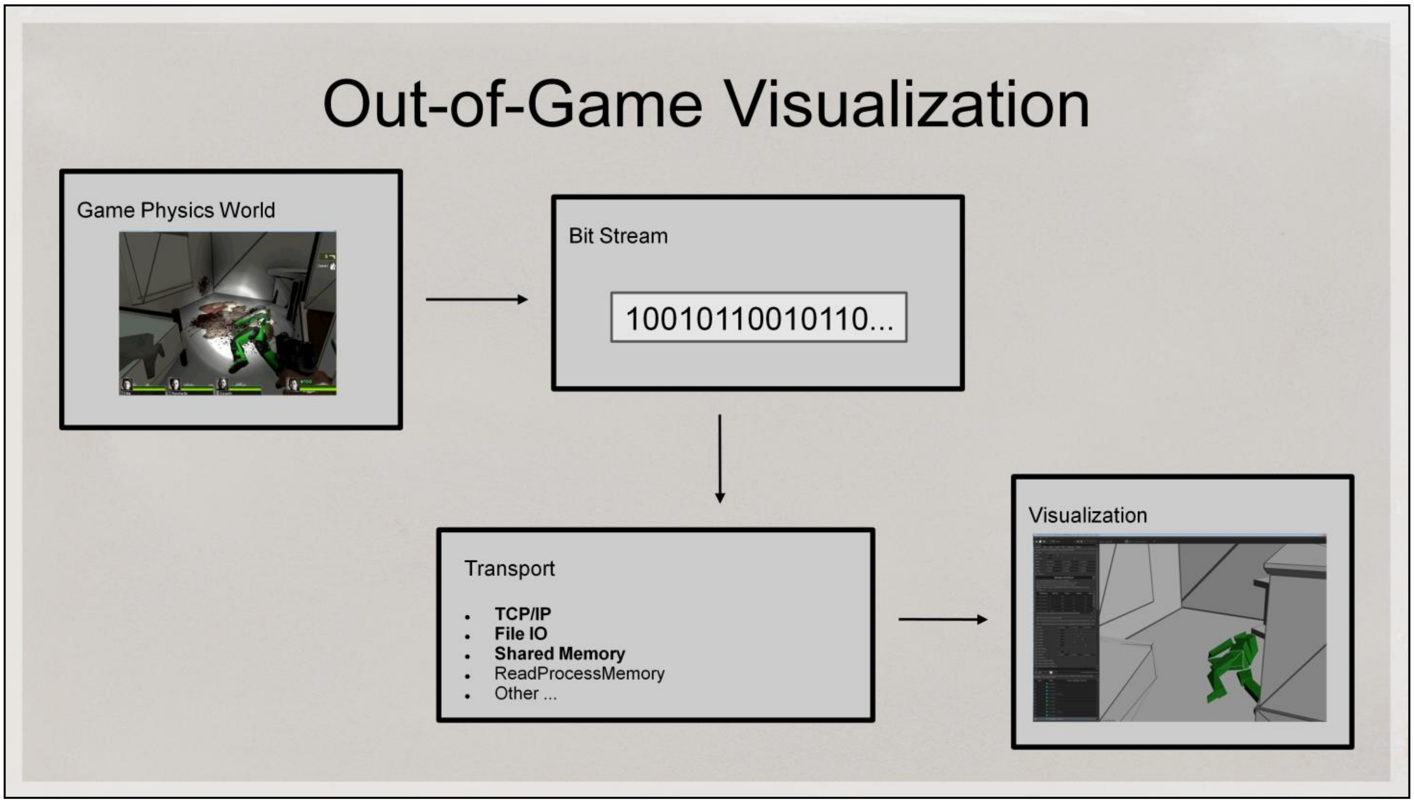
External Object

*** Published to snooper

A word about organizing your data strucutres.

It's most convenient to have a root object from which you can crawl all your data.
I just gather all the globals in one object, and only publish a pointer to that one object to the snooper.

Physics will have some data structures. You'll parse and process them.
You'll probably have some external pointers, like pointers to the vertex and index buffers for debug drawing.
You can write routines for snooping and serialization of those things manually



Out-of-Game Visualization

Game Physics World

Bit Stream

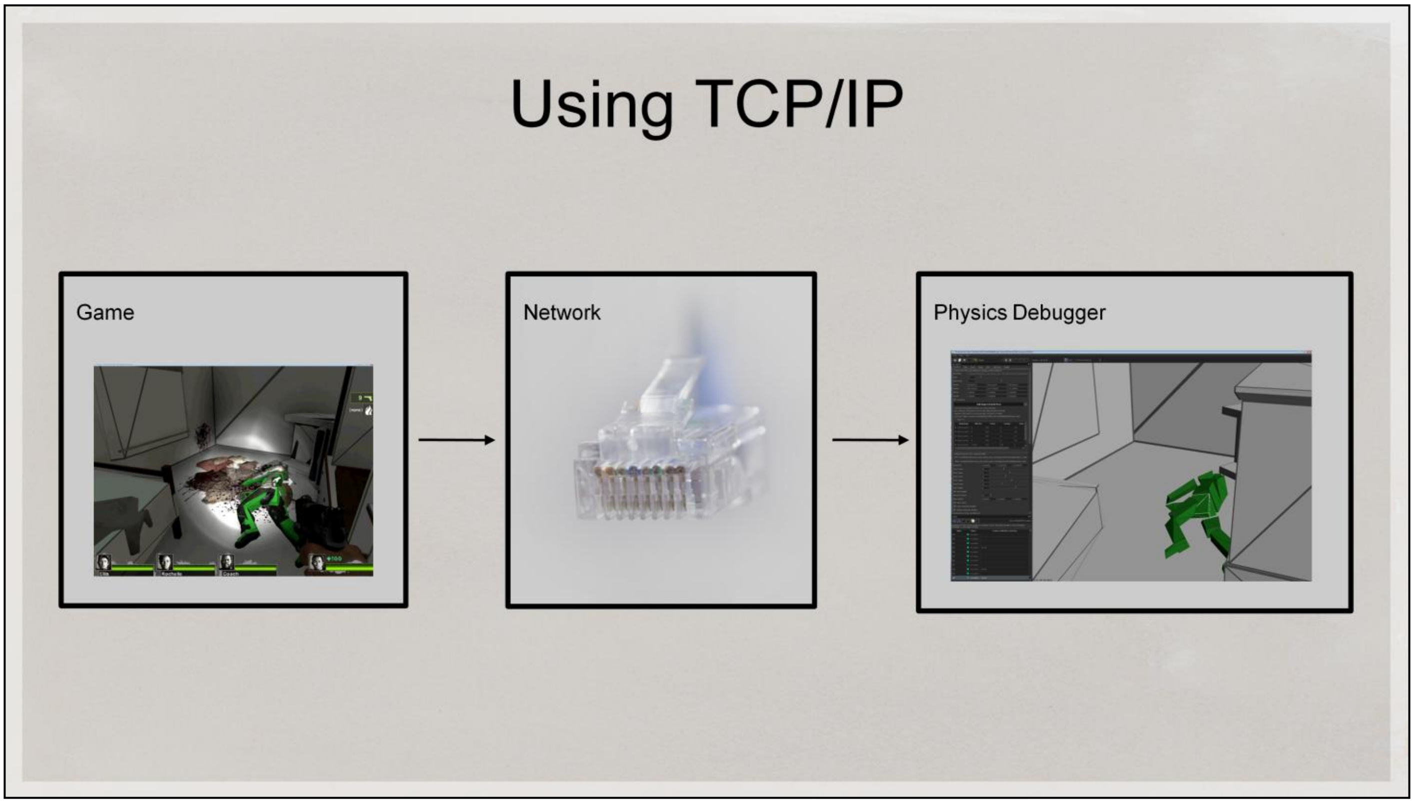
10010110010110...

Transport

- **TCP/IP**
- **File IO**
- **Shared Memory**
- ReadProcessMemory
- Other ...

Visualization

Another thought. Visualizing in an external app doesn't have to use ReadProcessMemory API.



Using TCP/IP

Game

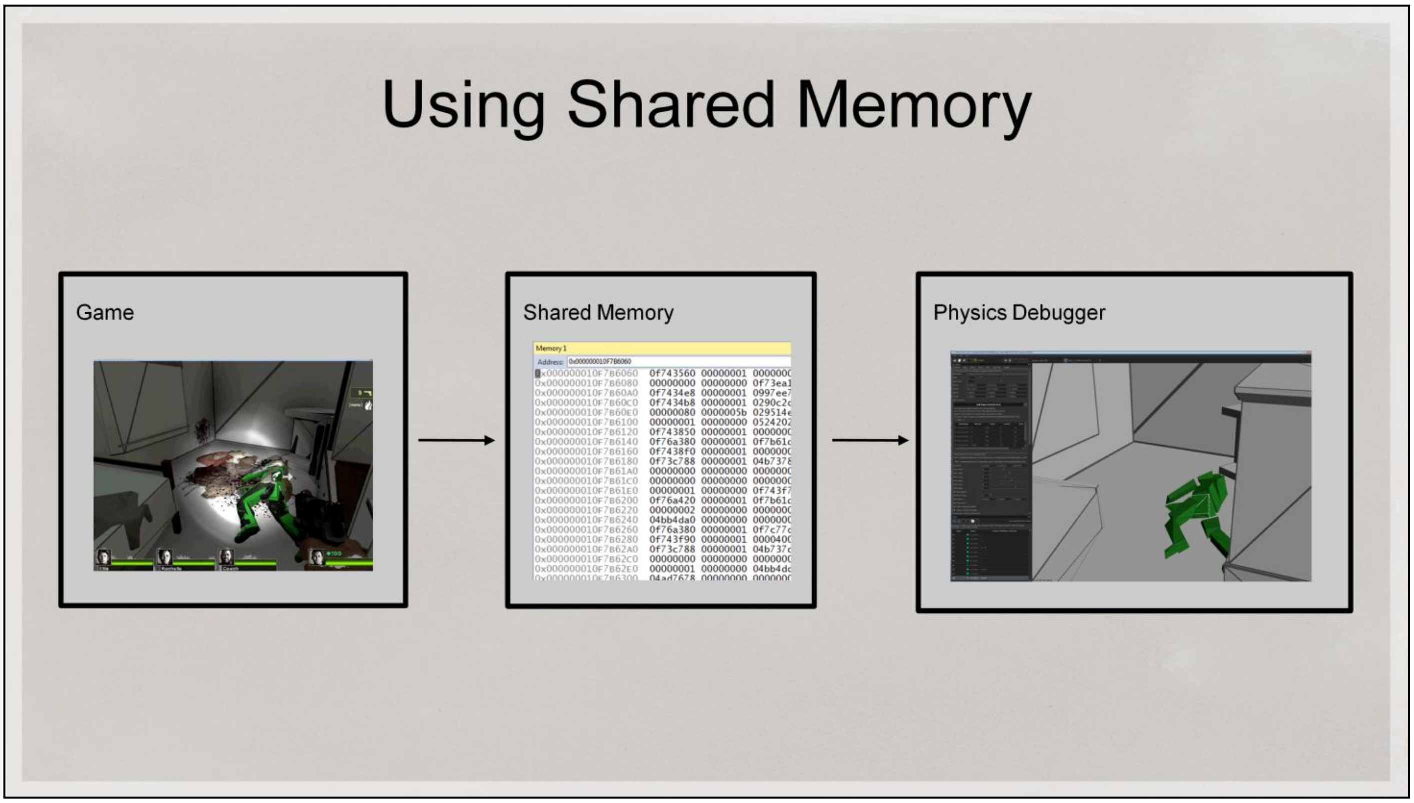
Network

Physics Debugger

All the leading physics engine have some sort of over-TCP/IP visualization.

Unfortunately it doesn't work when the game is stopped in debugger, frozen or crashed. Unless you stream your game state all the time, and that spends CPU. If you don't know where the problem is, and stream a lot of detailed data, it really takes a lot of CPU. And you have to remember to turn it on.

But it works on remote machine. So it's a viable option.



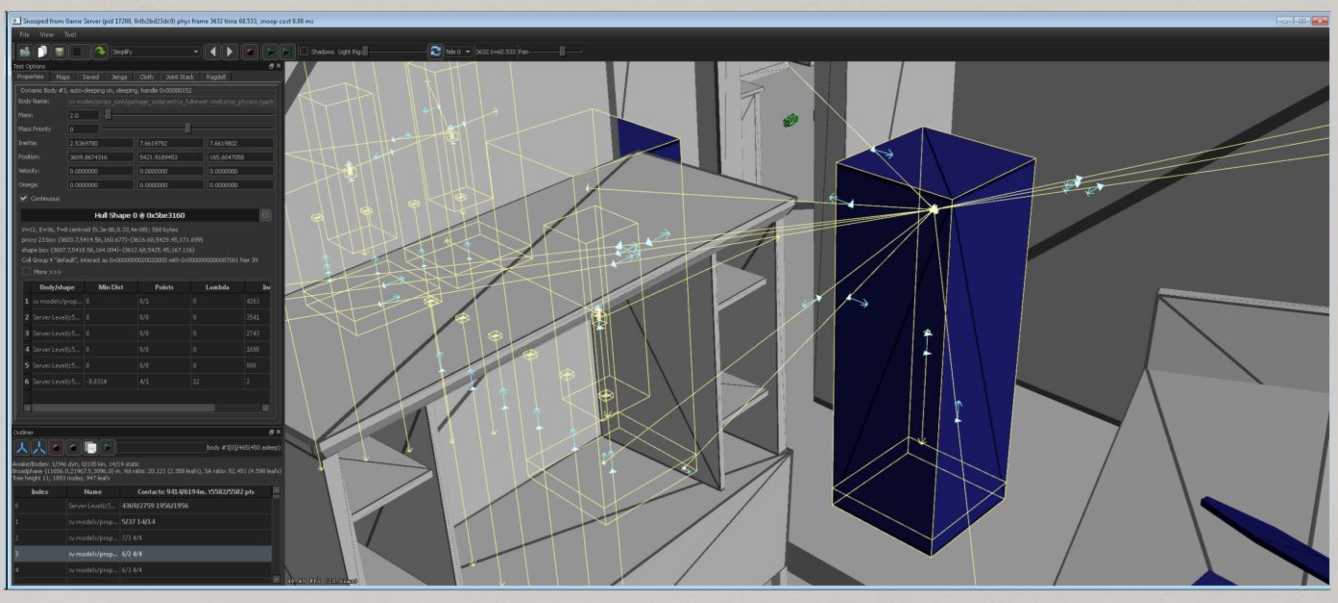
Using Shared Memory

```
0x000000010F7B6060   0f743560 00000001 0000000
0x000000010F7B6080   00000000 00000000 0f73ea1
0x000000010F7B60A0   0f7434e8 00000001 0997ee7
0x000000010F7B60C0   0f7434b8 00000001 0290c2c
0x000000010F7B60E0   00000080 0000005b 029514e
0x000000010F7B6100   00000001 00000000 0524202
0x000000010F7B6120   0f743850 00000001 0000000
0x000000010F7B6140   0f76a380 00000001 0f7b61c
0x000000010F7B6160   0f7438f0 00000001 0000000
0x000000010F7B6180   0f73c788 00000001 04b7378
0x000000010F7B61A0   00000000 00000000 0000000
0x000000010F7B61C0   00000000 00000000 0000000
0x000000010F7B61E0   00000001 00000000 0f743f7
0x000000010F7B6200   0f76a420 00000001 0f7b61c
0x000000010F7B6220   00000002 00000000 0000000
0x000000010F7B6240   04bb4da0 00000000 0000000
0x000000010F7B6260   0f76a380 00000001 0f7c77c
0x000000010F7B6280   0f743f90 00000001 000040c
0x000000010F7B62A0   0f73c788 00000001 04b737c
0x000000010F7B62C0   00000000 00000000 0000000
0x000000010F7B62E0   00000001 00000000 04bb4dc
0x000000010F7B6300   04ad7678 00000000 0000000
```

An alternative to TCP/IP that works when the game is stopped in debugger (or frozen or crashed) would be to use shared memory.

Win32 API allows processes to share memory directly. One way is to use a memory-mapped file, but you need to dedicate a region of memory for sharing, and some of the memory would be passed down from the game (like pointer to collision resources), and it would be very inconvenient to make sure all of that is allocated in the shared memory region

Game-centric Viz



Snooped from Game Server (pid 17200, 0x0x2bd23dc0) phys frame 3632 time 60.533, snoop cost 0.80 ms

3632.t=60.533

Dynamic Body #3, auto-sleeping on, sleeping, handle 0x00000152

Body Name:	sv models/props_junk/garbage_sodacan01a_fullsheet.vtx@prop_physics:/garb		
Mass:	2.0		
Mass Priority	0		
Inertia:	2.5369780	7.6619792	7.6619802
Position:	3609.8674316	5421.9189453	165.6047058
Velocity:	0.0000000	0.0000000	0.0000000
Omega:	0.0000000	0.0000000	0.0000000

☑ Continuous

Hull Shape 0 @ 0x5be3160

V=12, E=36, F=8 centroid (5.3e-06,0.33,4e-08); 568 bytes
proxy 23 box (3603.7,5414.56,160.677)-(3616.68,5429.45,171.699)
shape box (3607.7,5418.56,164.094)-(3612.68,5425.45,167.116)
Coll Group 4 "default", interact as 0x0000000020020000 with 0x0000000000087001 hier 39

□ More >>>

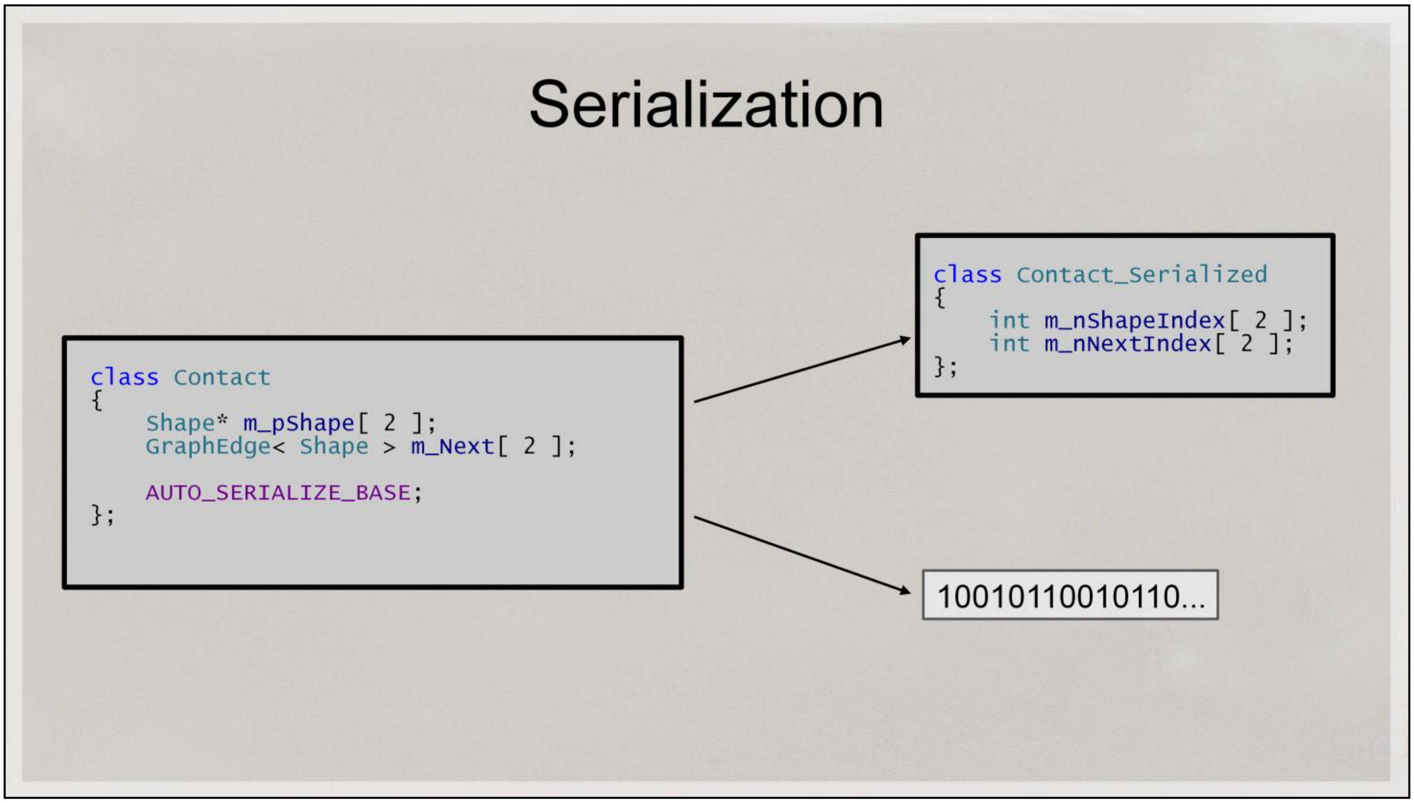
	Body/shape	Min Dist	Points	Lambda	In
1	sv models/prop...	0	0/1	0	4283
2	Server Level(c5...	0	0/0	0	3541
3	Server Level(c5...	0	0/0	0	2743
4	Server Level(c5...	0	0/0	0	1698
5	Server Level(c5...	0	0/0	0	909
6	Server Level(c5...	-0.0314	4/1	12	2

body #3[0]/465(450 asleep)

Awake/Bodies: 1/346 dyn, 0/105 kin, 14/14 static
Broadphase (11656.0,21967.5,3096.0) in. Vol ratio: 20.123 (2.358 leafs), SA ratio: 52.451 (4.598 leafs)
Tree height 11, 1893 nodes, 947 leafs

Index	Name	Contacts: 9414/6194m, t5582/5582 pts
0	Server Level(c5...	4369/2759 1956/1956
1	sv models/prop...	5/37 14/14
2	sv models/prop...	7/3 4/4
3	sv models/prop...	6/2 4/4
4	sv models/prop...	6/3 4/4

- Hopefully we'll eventually be able to save/load/rewind the whole game with this method. Our game code base is much bigger than physics engine though, so that's a lot of work.



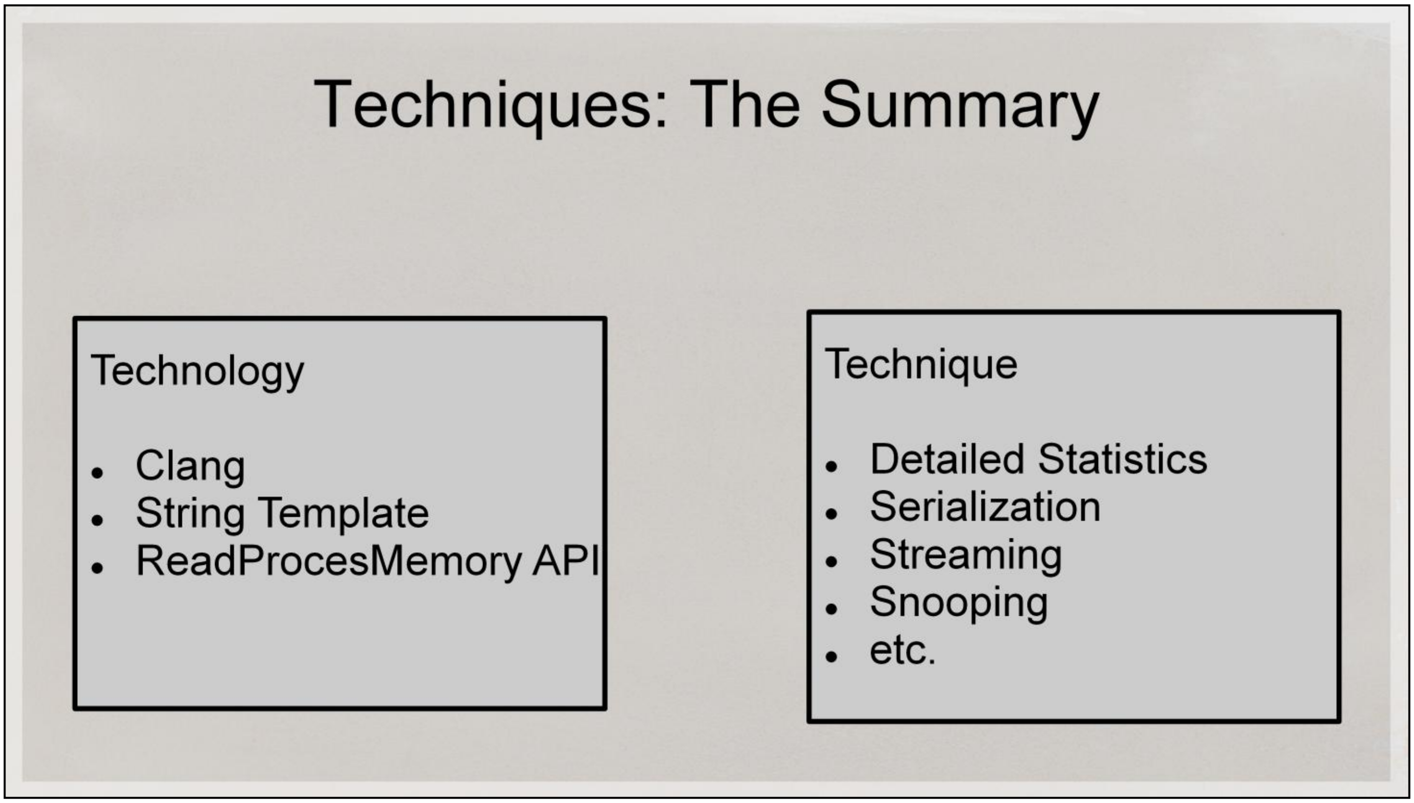
Serialization

```
class Contact
{
    Shape* m_pShape[ 2 ];
    GraphEdge< Shape > m_Next[ 2 ];

    AUTO_SERIALIZE_BASE;
};
```

```
class Contact_Serialized
{
    int m_nShapeIndex[ 2 ];
    int m_nNextIndex[ 2 ];
};
```

10010110010110...

- We are using the same framework to generate serialize/unserialize code. Every time we change our data structures, we just re-run the parser/generator, and it's all updated
- We serialize/unserialize a byte stream. But another variant would be e.g. to generate a reflected version of all your data, with indices instead of pointers. If you have reflection API, it will let you version the serialized data



Techniques: The Summary

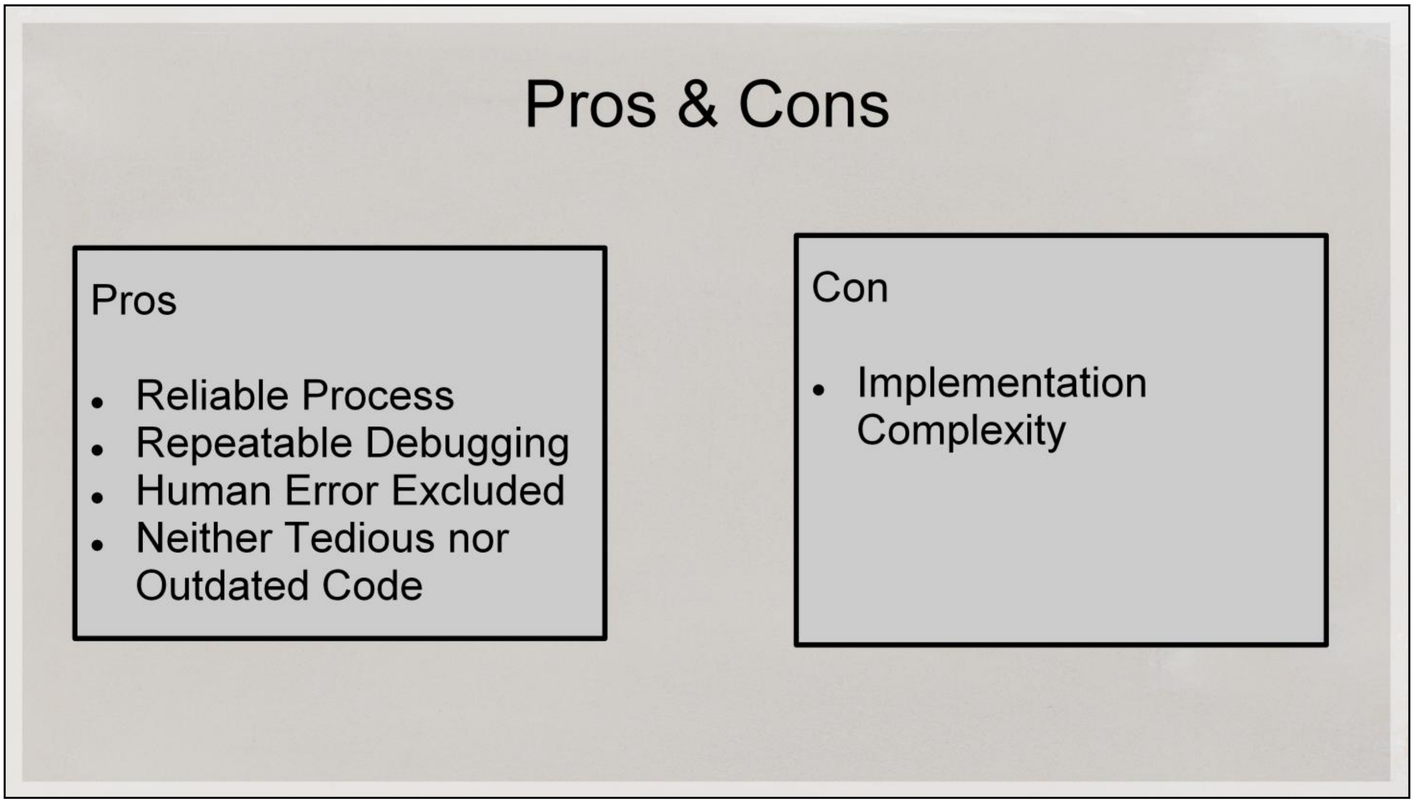
Technology

- Clang
- String Template
- ReadProcesMemory API

Technique

- Detailed Statistics
- Serialization
- Streaming
- Snooping
- etc.

- To summarize, we were talking about a few uses of two technologies: Clang and String Template, and one simple Windows API: ReadProcessMemory.
- Together, these technologies can enable very powerful and complete serialization, an interesting debugging technique I call snooping, gathering detailed statistics without tedious coding and without that code going stale the next time someone adds a new array somewhere.
- The same serialization template can also be used to send data over TCP/IP, and/or serialize into an alternative data structure with reflection API (e.g. into a series of dictionaries) that can be saved and loaded into newer versions of the engine, and many more uses.
- It is possible to use snooping to read and display every frame in your game
- You will probably skip some frames unless you take care to synchronize
- You will probably waste time at the mutex, for the game has to wait for the snooper to finish, and it's less efficient to snoop than to pack local game data and send it through a pipe or socket.
- It is much faster to let the game actively serialize the world delta every frame and send it over the wire. It will also guarantee no skipped frames.
- It is easier to just reuse existing serialization and send every frame over the wire
- For a quick fix, just see the biggest pieces of data (probably mesh descriptions) and cut them out of the stream if you already sent them
- When visualizing streamed physics, I tied the Vbs/IBs directly into the world data structures.
- This automatically recreated Vbs/IBs on every streamed frame
- To fix that, I had to cache those debug objects off and have special callbacks reuse them. I did it with clang's annotations.



Pros & Cons

Pros

- Reliable Process
- Repeatable Debugging
- Human Error Excluded
- Neither Tedious nor Outdated Code

Con

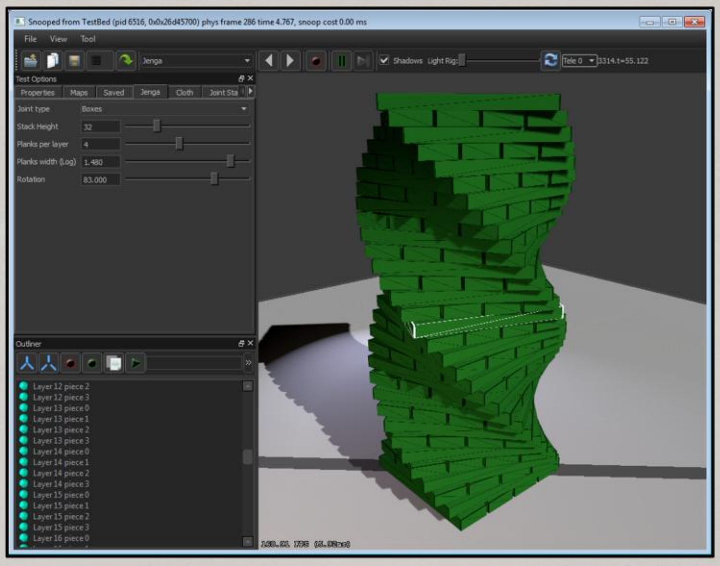
- Implementation Complexity

- Full-world serialization of this type includes everything, and has very high reliability. You don't have to worry about missing a thing or two: the parser/generator is unlikely to make a human error.
- This is usable to dump suspicious game frames for later examination, or streaming them. When debugging later, it's very much like debugging in-game, but it's repeatable and much more convenient.
- I don't like manual non creative work, and serialization code is always very low in creativity. Getting rid of it makes iteration faster and doesn't distract me from the creative process.
- Among the drawbacks, this method is more complex to start using it. You can still use the old methods in parallel with this method.

Conclusion

Make work fun! It pays to do that.

Q&A



Snooped from TestBed (pid 6516, 0x0x26d45700) phys frame 286 time 4.767, snoop cost 0.00 ms

Joint type	Boxes
Stack Height	32
Planks per layer	4
Planks width (Log)	1.480
Rotation	83.000

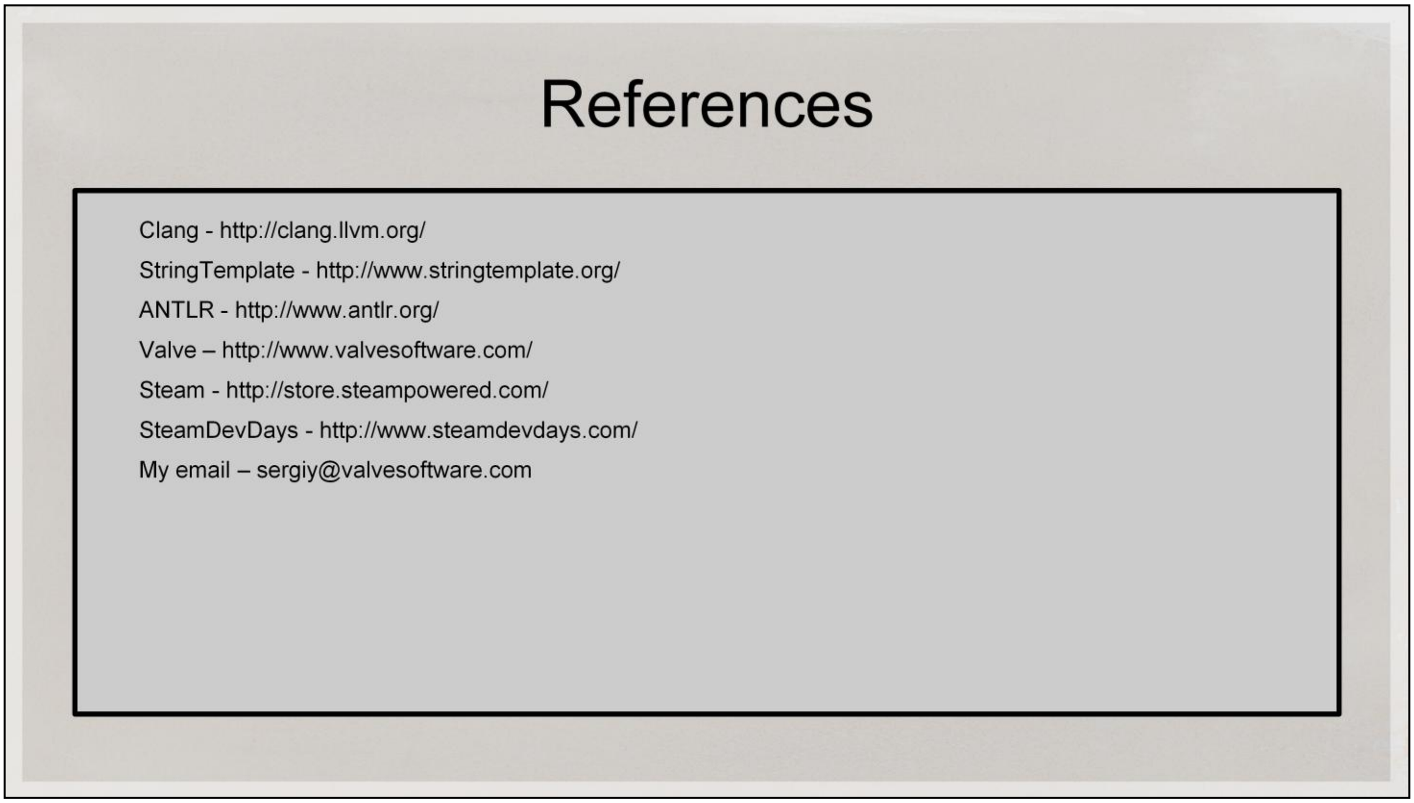
Layer 12 piece 2
Layer 12 piece 3
Layer 13 piece 0
Layer 13 piece 1
Layer 13 piece 2
Layer 13 piece 3
Layer 14 piece 0
Layer 14 piece 1
Layer 14 piece 2
Layer 14 piece 3
Layer 15 piece 0
Layer 15 piece 1
Layer 15 piece 2
Layer 15 piece 3
Layer 16 piece 0

Special Thanks: Content & Delivery

Erwin Coumans
Dirk Gregorius
Dennis Gustafsson
Julien Merceron

Special Thanks: Art & Style

Ricardo Ariza
Jason Brashill
Cam Fielding
Tristan Reidford



References

Clang - http://clang.llvm.org/

StringTemplate - http://www.stringtemplate.org/

ANTLR - http://www.antlr.org/

Valve – http://www.valvesoftware.com/

Steam - http://store.steampowered.com/

SteamDevDays - http://www.steamdevdays.com/

My email – sergiy@valvesoftware.com