ShaderSense: A Visual Studio Language Extension

Garrett Kiel, Cory Luitjohan, Michael Covert, Edward Han, Phil Slama, Feng Cao

Table of Contents

Project Description

ShaderSense is a language extension for Visual Studio that provides Intellisense features for High Level Shader Language. The Intellisense features include tooltips, auto-completions, member completions and parameter info. Additionally, it also does some amount checking for syntax errors, and highlights these errors with a red wavy underline. Auto-completion is a list of items that pops up during coding that a person can select from to quickly insert into the code. Member completion is a list of items that pops up after a person types a period following a structure variable, and this list contains the members of that structure type. Parameter info gives a tooltip-like box after typing an open parenthesis following a function name which shows the function signature and highlights the current parameter that the programmer should enter.

Process

We used XP based around manual testing rather than automatic testing. As we finished writing a feature, we could easily see if the feature worked or not because the results of the feature were all very visible. It was easy to check that syntax coloring worked because we can see the different colors of the text and make sure the correct tokens were the correct color. It was easy to check to see that auto-completions were working because we could just scroll through the list that Visual Studio showed us and make sure everything was in the list that we expected to be in the list. Member completions were easy to check because we could just look at the list of members that Visual Studio showed us and make sure that those were in fact the members of that structure type. We could check parameter info by seeing if the function signature that Visual Studio showed matched that of the actual function. We could check to see if the red underlining worked because we knew which tokens should be underlined and we could see if there was a red underline under those tokens.

Throughout the project, we were always doing pair programming. For the entire project, Phil and Feng were working on getting some form of automated testing working. In the beginning, Michael and Garrett worked on getting as much of the HLSL grammar written as possible, and Ed and Cory worked on the getting the first Intellisense feature of tooltips working. After that, Cory and Garrett worked on auto-completions and member completions, and Ed and Michael worked on parameter info. Working in pairs most likely helped us write better, working code the first time because there was always someone watching out for coding errors, and there were always two people working together to try and figure out how to implement each feature.

We worked on these features in iterations. During each iteration, we would try and get another feature working. Additionally, we would sometimes go back and clean up our work from the previous iteration because some of that work could be relevant to the current iteration's work. We tried to plan out the iterations at the very beginning of the project, and for the most part, we managed to stay on track.

Feng and Phil spent a lot of time working on getting automated testing working and had their own process for how to test things. The most effective way to test Syntax Highlighting was to compare the attributes of each token against its "correct" value. The most efficient way of doing this was to output the token values into a text file and compare it against the "expected" output. Since the parser will not actually output token values until ShaderSense opens a file, we had to figure out how to have our plug-in automatically open and close files to generate the required token information. Because the process of outputting token info, opening/closing files, and writing token info takes time, we had to manually approximate the wait times to ensure a consistent test result was returned. Testing tooltips was similar to test Syntax Highlighting in that rather than getting the token values of each token, we are now getting the "token descriptions” and comparing those values instead. Testing for member completion/auto-completion automatically does not seem practical. The process requires dynamic user interaction, so the only way to test correct outputs of auto-completion/member completion is to send key strokes to the visual studio plug in. But the process seems redundant and needlessly complicated since the coder that uses the system can test the process much more effectively.

Requirements & Specifications

We have the following use cases for our project:

Display auto-completions:

Primary Actor: System
Goal in context: Displaying keywords or other names automatically during typing, or when prompted by the user.
Scope: Visual Studio Language Extension
Level: Subfunction
Stakeholders:
Programmer: Wants an easy way to see the available keywords.
Precondition: None
Guarantees: Displays the keywords, etc. that begin with the typed letters, if there are any.
Triggers: Programmer begins typing a few characters, or programmer selects the Display auto-completions option.
Success condition: The appropriate keywords, etc. are displayed and can be selected by the user.
Extensions: None

Display member completions:

Primary Actor: System
Goal in context: Displaying members of an object based on the object's type.
Scope: Visual Studio Language Extension
Level: Subfunction
Stakeholders:
Programmer: Wants an easy way to see the available members for a certain object.
Precondition: The name preceding the period is an object.
Guarantees: Displays the members of that object, if there are any.
Triggers: Programmer types a period following an object's name.
Success condition: The appropriate members are displayed and can be selected by the user.
Extensions: None

Lex the HLSL file:

Primary Actor: Language Service

Goal in context: Convert a sequence of characters into a series of tokens

Scope: Visual Studio Language Extension

Level: Subfunction

Stakeholders:

Language Service: The language service needs this case to complete successfully in order to perform its function.

Precondition: The language service must be registered with Visual Studio and associated with HLSL source files.

Guarantees: It reads through a string of characters and determines the tokens in the input.

Triggers: Visual Studio invokes the lexer.

Extensions: None

Respond to Mouse-Over Events:

Primary Actor: Visual Studio

Goal in context: Provide tool tips when the user moves the mouse over the code.

Scope: Visual Studio Language Extension

Level: User

Stakeholders:

Language Service: The language service needs this case to know when to provide information.

Visual Studio: Visual Studio uses this to tell the Language Service when to generate information.

Programmer: The programmer needs this functionality to quickly and easily see information about the parts of the code.

Precondition: The language service must be registered with Visual Studio and associated with HLSL source files. The file the tool tip is getting information about must be a HLSL source file. The source code must also successfully parse up to where the mouse is over.

Guarantees: None.

Triggers: The user moves the mouse over a part of some HLSL source code.

Extensions:

- If the source code fails to lex or parse, the use case fails and no tool tip is displayed.
- If the language service is unable to obtain information about the item the mouse is over, the use case fails and no tool tip is displayed.

<u>Display parameter info:</u>

Primary Actor: Visual Studio

Goal in context: Display the parameter info of the function call the user just typed.

Scope: Visual Studio Language Extension

Level: Subfunction

Stakeholders:

Programmer: The programmer wants to quickly and easily know the parameters to the function he is trying to use.

Precondition: The source file was correctly and completely parsed.

Guarantees: None.

Triggers: The programmer typed an open parenthesis immediately following a function name.

Success condition: The parameter info for the function is displayed correctly, and as parameters are entered, the following parameter gets highlighted.

Extensions:

- If the source code failed to parse the function, the use case fails and no parameter info is displayed.

<u>Red underline syntax errors:</u>

Primary Actor: Visual Studio

Goal in context: Display red underlines beneath identifiers in the code that aren't recognized and may cause syntax errors when the shader is compiled.

Scope: Visual Studio Language Extension

Level: Subfunction

Stakeholders:

Programmer: The programmer wants to be alerted to possible syntax errors as quickly as possible when writing code.

Precondition: The source file was completely parsed.

Guarantees: None.

Triggers: Visual studio periodically calls for a syntax check parse.

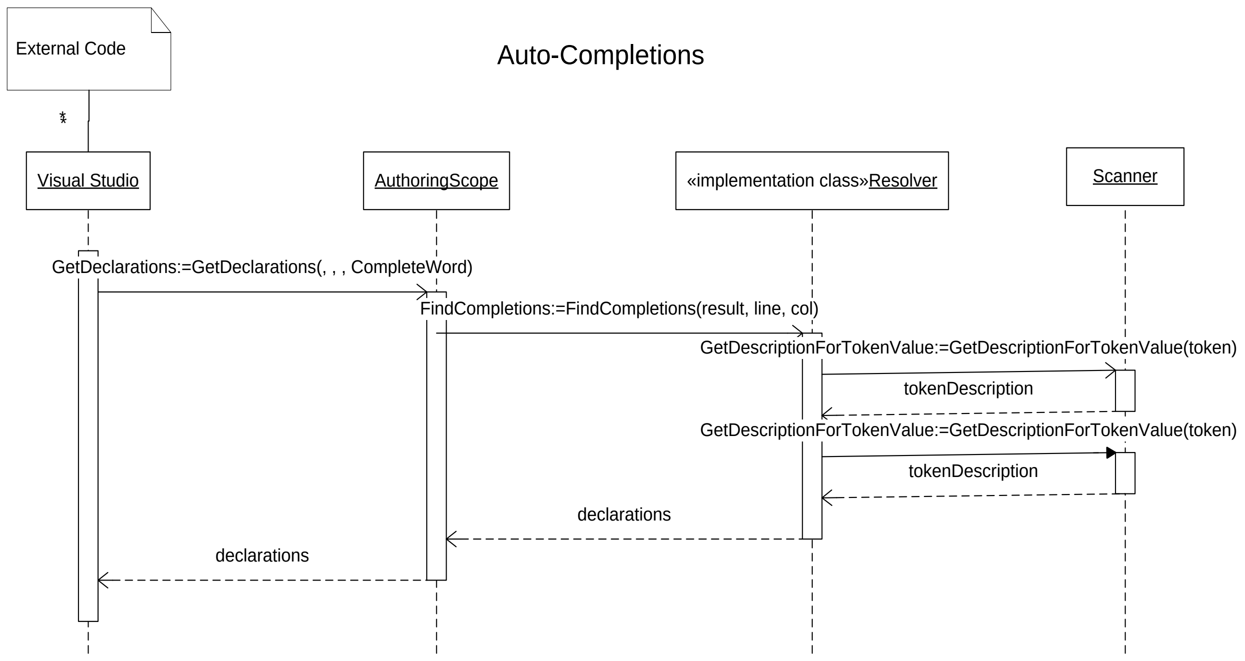
Success condition: Identifiers that would cause syntax errors during compilation have red wavy underlines beneath them.

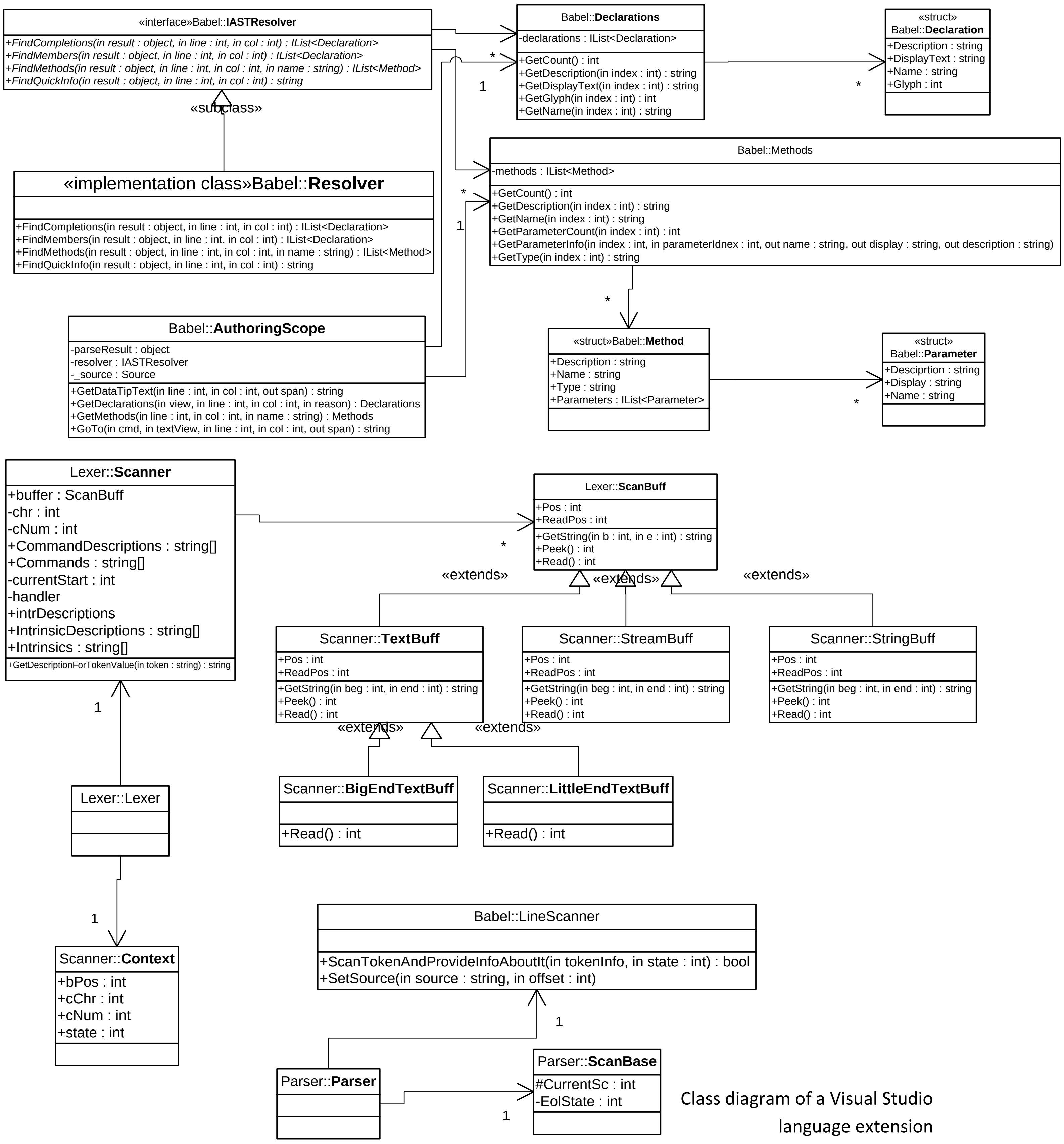
Extensions: None.

Architecture & Design

Much of the architecture and design of our project was based off of the code that was provided by the Visual Studio SDK and the methods that Visual Studio exposed to us for using. In addition to this, we built our language service off of the Managed Babel Language Service framework. These two constraints largely decided what our classes' responsibilities were and what methods needed to be in them. We added a few additional methods, but these methods were all called at some point in the pre-defined methods. The UML class diagram on the next page attempts to detail the design of a Visual Studio language service.

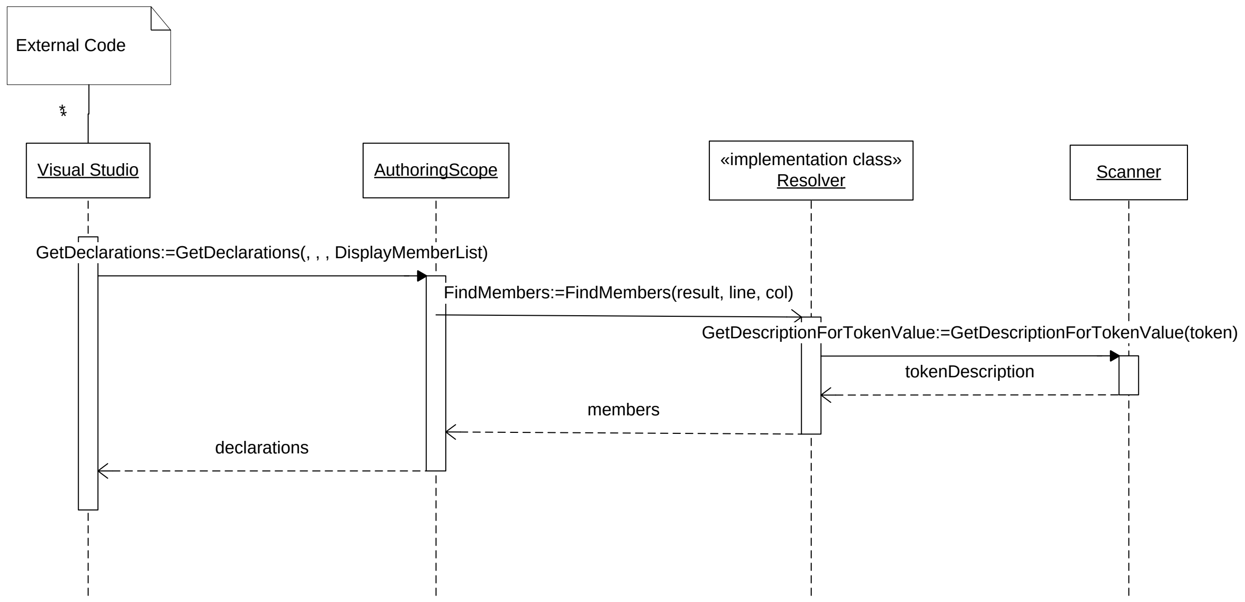
Our design for auto-completions is fairly straightforward. We compile a list of keywords, intrinsic functions, function names, and variables that are in scope, and give this list back to Visual Studio to display to the user. The sequence diagram below shows the main classes that are involved in the order of events that occur during this process.


External Code
Auto-Completions
Visual Studio
AuthoringScope
«implementation class»Resolver
Scanner
GetDeclarations:=GetDeclarations(, , , CompleteWord)
FindCompletions:=FindCompletions(result, line, col)
GetDescriptionForTokenValue:=GetDescriptionForTokenValue(token)
tokenDescription
GetDescriptionForTokenValue:=GetDescriptionForTokenValue(token)
tokenDescription
declarations
declarations


«interface»Babel::IASTResolver
+FindCompletions(in result : object, in line : int, in col : int) : IList<Declaration>
+FindMembers(in result : object, in line : int, in col : int) : IList<Declaration>
+FindMethods(in result : object, in line : int, in col : int, in name : string) : IList<Method>
+FindQuickInfo(in result : object, in line : int, in col : int) : string
«subclass»
«implementation class»Babel::Resolver
+FindCompletions(in result : object, in line : int, in col : int) : IList<Declaration>
+FindMembers(in result : object, in line : int, in col : int) : IList<Declaration>
+FindMethods(in result : object, in line : int, in col : int, in name : string) : IList<Method>
+FindQuickInfo(in result : object, in line : int, in col : int) : string
Babel::AuthoringScope
-parseResult : object
-resolver : IASTResolver
-_source : Source
+GetDataTipText(in line : int, in col : int, out span) : string
+GetDeclarations(in view, in line : int, in col : int, in reason) : Declarations
+GetMethods(in line : int, in col : int, in name : string) : Methods
+GoTo(in cmd, in textView, in line : int, in col : int, out span) : string
Babel::Declarations
-declarations : IList<Declaration>
+GetCount() : int
+GetDescription(in index : int) : string
+GetDisplayText(in index : int) : string
+GetGlyph(in index : int) : int
+GetName(in index : int) : string
*
1
«struct»
Babel::Declaration
+Description : string
+DisplayText : string
+Name : string
+Glyph : int
*
Babel::Methods
-methods : IList<Method>
+GetCount() : int
+GetDescription(in index : int) : string
+GetName(in index : int) : string
+GetParameterCount(in index : int) : int
+GetParameterInfo(in index : int, in parameterIdnex : int, out name : string, out display : string, out description : string)
+GetType(in index : int) : string
*
1
*
«struct»Babel::Method
+Description : string
+Name : string
+Type : string
+Parameters : IList<Parameter>
«struct»
Babel::Parameter
+Desciprtion : string
+Display : string
+Name : string
*
Lexer::Scanner
+buffer : ScanBuff
-chr : int
-cNum : int
+CommandDescriptions : string[]
+Commands : string[]
-currentStart : int
-handler
+intrDescriptions
+IntrinsicDescriptions : string[]
+Intrinsics : string[]
+GetDescriptionForTokenValue(in token : string) : string
*
Lexer::ScanBuff
+Pos : int
+ReadPos : int
+GetString(in b : int, in e : int) : string
+Peek() : int
+Read() : int
«extends»
«extends»
«extends»
Scanner::TextBuff
+Pos : int
+ReadPos : int
+GetString(in beg : int, in end : int) : string
+Peek() : int
+Read() : int
Scanner::StreamBuff
+Pos : int
+ReadPos : int
+GetString(in beg : int, in end : int) : string
+Peek() : int
+Read() : int
Scanner::StringBuff
+Pos : int
+ReadPos : int
+GetString(in beg : int, in end : int) : string
+Peek() : int
+Read() : int
«extends»
«extends»
Scanner::BigEndTextBuff
+Read() : int
Scanner::LittleEndTextBuff
+Read() : int
1
Lexer::Lexer
1
Scanner::Context
+bPos : int
+cChr : int
+cNum : int
+state : int
Babel::LineScanner
+ScanTokenAndProvideInfoAboutIt(in tokenInfo, in state : int) : bool
+SetSource(in source : string, in offset : int)
1
Parser::Parser
Parser::ScanBase
#CurrentSc : int
-EolState : int
1
Class diagram of a Visual Studio language extension

Our design for member completions may actually be simpler than auto-completions. It looks up the type of structure that the variable is, and then returns a list of members of that type that the parser had parsed.

Member Completions


External Code
*
Visual Studio
AuthoringScope
«implementation class» Resolver
Scanner
GetDeclarations:=GetDeclarations(, , , DisplayMemberList)
FindMembers:=FindMembers(result, line, col)
GetDescriptionForTokenValue:=GetDescriptionForTokenValue(token)
tokenDescription
members
declarations

A metaphor for our project could be a company. The boss would be the external Visual Studio code that tells the employees to do some work. There are some employees that do the parsing work, and some employees that are the resolver and authoring scope who do work on the results of the parsing employees' work. Then the employees tell the boss the results of their work, and the boss presents the results to the customer.

Future Plans

Currently, our future plans for this project are to put it on Google Code for anybody in the group who wants to continue working on it.

Personal Reflections

Garrett:

Due to the fact that this project was my idea, I of course enjoyed working on this project. I learned a lot about how Visual Studio works with extensions. I learned that there is a lot of magic that Visual Studio does on its own which makes it somewhat difficult to figure out how everything works together. Also, it greatly helped that I had already taken CS421 in a previous semester because we had to write a lexer and parser for HLSL, and we learned how to do that in CS421. My original reason for wanting to do a project like this was because I wanted to write shaders for Half-Life 2 mods, but I didn't really know HLSL very well. Doing this project has helped me learn the HLSL language somewhat, as well as just generally making it a little easier to write HLSL code with all these Intellisense features we implemented.

Michael:

I really enjoyed working on this project. I gained a lot of experience working with plugging in to other people's code; it was quite different from my experience with Eclipse last semester. Plugging into Eclipse seemed much smoother and seamless, but plugging into Visual Studio was, at times, a pain. The main reason for this, I think, is that Eclipse provided source code, and Visual Studio did not. This made it easier to tell what was going on behind the scenes in Eclipse. I think that my experience working with plugging into non-open source code will help out a lot in the future, and I'm glad to have gotten this experience.

I also learned a lot about HLSL and about parsing and lexing and how they interact with an IDE as we made the parser and lexer. I had never worked with HLSL before this, so I was able to learn a lot about the language as we worked. I've only worked briefly with parsers before, and thus I learned a lot more about the parser as well.

Cory:

I enjoyed and learned a lot from this project. I feel like it went pretty well. I learned a great deal about writing parsers and what Visual Studio has to do in order to have good Intellisense. I learned a lot about how Intellisense actually works and the lexing and parsing that goes with it. I developed a good amount of skills reading documentation and making sense out of them. A great deal of this project was spent looking through documentation and trying to figure out how it related to our project. This skill could be very useful in the future. It was a huge pain a lot of times, but we delved through and accomplished our goals. I enjoyed working with our group. I knew all but two of them before but it was enjoyable to meet new people and incorporating them to the group.

Edward:

I feel that this project went pretty well, and I learned a great deal from it. We accomplished what we originally set out to do, which was to create a Visual Studio extension that helped make writing HLSL code easier. The extension extracts information from a source file and provide the user with various useful information. There were a few issues during development, but we usually found a solution to the problems.

I learned several things on this project. First, the project really highlighted the need for proper examples in frameworks. We had a great deal of understanding how parts of the framework works when there was no example. Aside from that, I learned a great deal about HLSL, something which I wanted to do eventually. I also learned several things about how Visual Studio works, which let me understand some of the issues that show up when I use the IDE.

Feng:

Working on Shadersense had being a valuable learning experience. It refreshed my memory on the RegEx grammar and allowed me to work closely with the parser and the lexer. I was able to see what the IDEs did behind the scenes for intellisense and doing so gave me a greater understanding of the system. Writing automated tests for Shadersense had being very helpful as well. Not only did I get to a chance to break down the system and identify what part of Shadersense did what, but I also learned things such as getting active copies of the Visual Studio project and sending commands to the active object automatically. Overall, working on Shadersense has been an interesting experience, and although I can’t see myself working on a project like this again in the near future, it had been a very challenging learning experience.

Phil:

Working on the ShaderSense project has been a wonderful learning experience. First, it was cool to see how Visual Studio works behind the scenes. It was also good to put the stuff I learned in CS 421 to practical use when designing the parser and lexer files. I really enjoyed working on automated testing, as this presented a different type of challenge than traditional coding. With automated testing, we had to find work arounds to inject our test code into the Visual Studio code to get output. Thus, the challenge was not getting a for loop to run in the way we expected, but rather learning how all the pieces of Visual Studio worked and how to get them to work for our project. I also learned the valuable lesson that not all projects warrant automated testing. It was easier for the developers in our project to test their own code with a simple click or press of a key than for my partner and I to write test cases. We spent far more time writing our test cases than it took for the developers to test our code by hand. Overall though, I think this project was very helpful in my development as a programmer.

Appendix

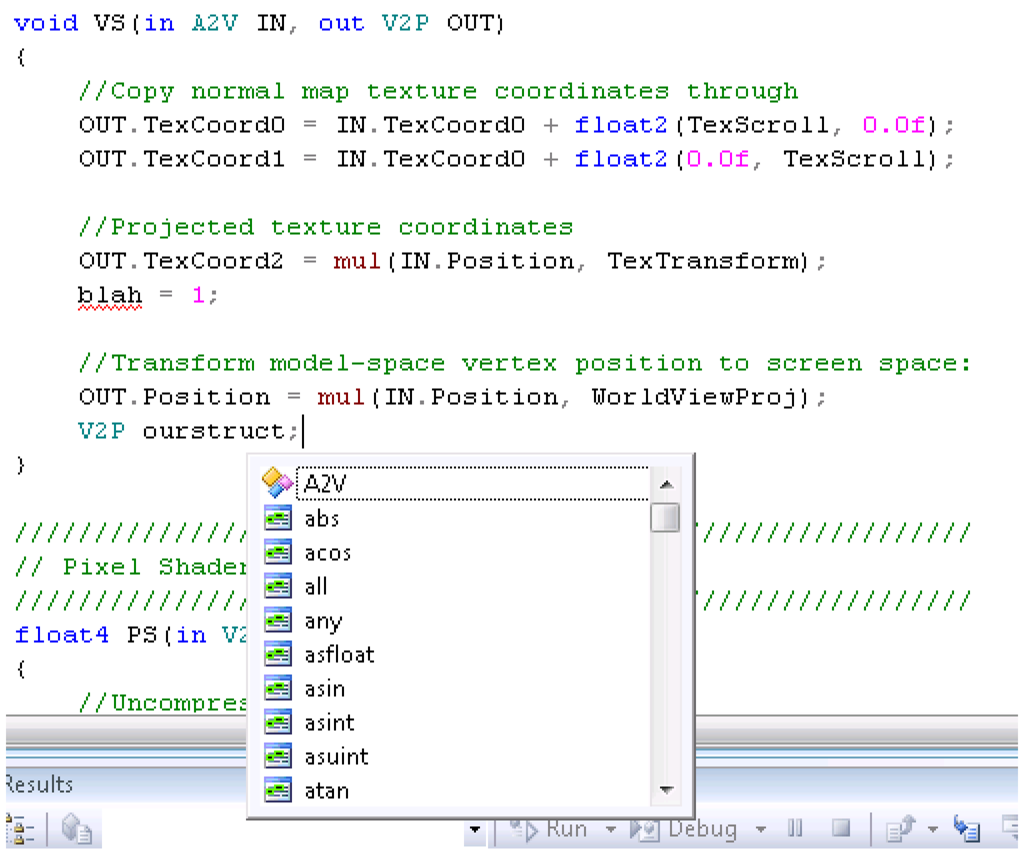
Installation:

Currently, we don't have an installer for the project, but we plan on following the instructions in the video, "How Do I: Deploy a Visual Studio Integration Package?" (http://msdn.microsoft.com/en-us/vstudio/cc627235.aspx). Once an installer is created, it should simply be a matter of running the installer.

To install and run the source code, you first need to install Visual Studio 2008, then Visual Studio 2008 SP1 (http://www.microsoft.com/downloads/details.aspx?FamilyID=fbee1648-7106-44a7-9649-6d9f6d58056e&DisplayLang=en), and then the Visual Studio 2008 SDK (http://www.microsoft.com/downloads/details.aspx?FamilyId=59EC6EC3-4273-48A3-BA25-DC925A45584D&displaylang=en). Then, check out the code from the repository. Then, you want to run Visual Studio in the experimental hive, and open ShaderSense.sln in there. Once you have opened up the solution, open the properties of the ShaderSense project and select the Debug tab. Change the Start Action to "Start external program" and browse to your Visual Studio install directory and find and select devenv.exe. Under "Command line arguments", you also need to put the following in (without the quotes): "/ranu /rootsuffix Exp".

How to run:

Running our extension is simply a matter of opening an HLSL shader file(a file with a .fx extension) in Visual Studio, and Visual Studio will automatically run our code. To get auto-completions to show up, hit Ctrl+Space. To get member completions to show up, type a period following a variable of a structure type. To get parameter info to show up, type an opening parenthesis following a function name.



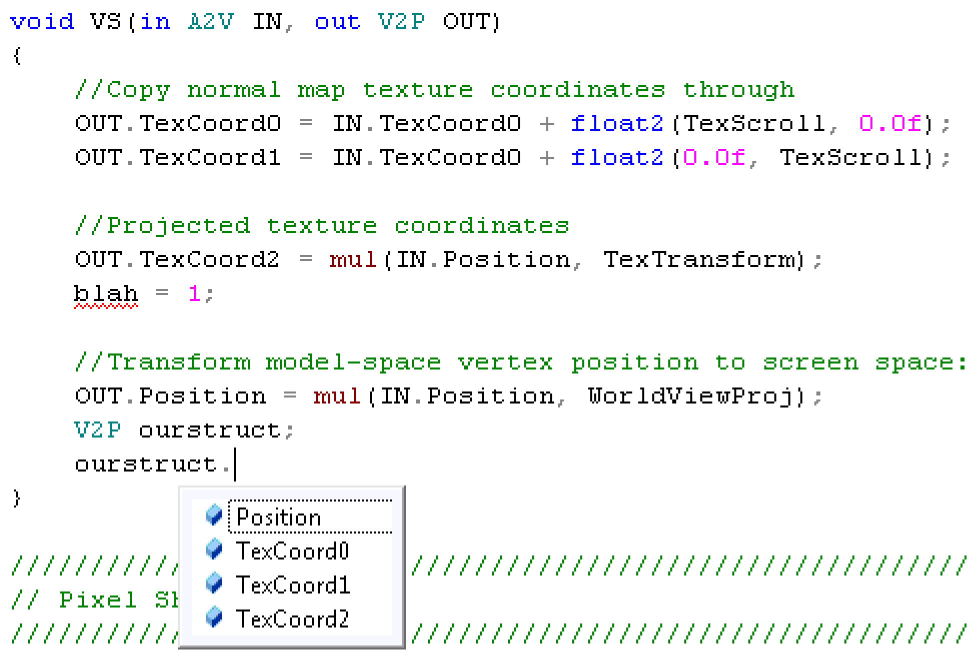
```
void VS(in A2V IN, out V2P OUT)
{
     //Copy normal map texture coordinates through
     OUT.TexCoord0 = IN.TexCoord0 + float2(TexScroll, 0.0f);
     OUT.TexCoord1 = IN.TexCoord0 + float2(0.0f, TexScroll);

     //Projected texture coordinates
     OUT.TexCoord2 = mul(IN.Position, TexTransform);
     blah = 1;

     //Transform model-space vertex position to screen space:
     OUT.Position = mul(IN.Position, WorldViewProj);
     V2P ourstruct;
}

/////////////////////////////////////////////////////////////////
// Pixel Shader
/////////////////////////////////////////////////////////////////
float4 PS(in V2
{
     //Uncompres
```

Auto-completions and red underlining



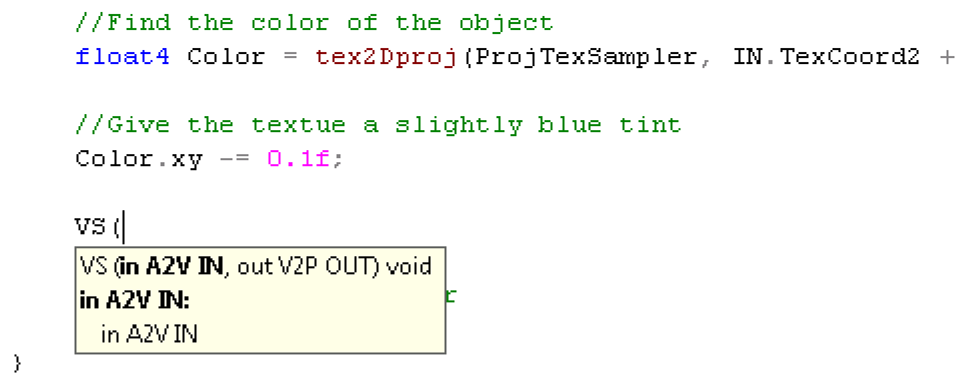
```
void VS(in A2V IN, out V2P OUT)
{
     //Copy normal map texture coordinates through
     OUT.TexCoord0 = IN.TexCoord0 + float2(TexScroll, 0.0f);
     OUT.TexCoord1 = IN.TexCoord0 + float2(0.0f, TexScroll);

     //Projected texture coordinates
     OUT.TexCoord2 = mul(IN.Position, TexTransform);
     blah = 1;

     //Transform model-space vertex position to screen space:
     OUT.Position = mul(IN.Position, WorldViewProj);
     V2P ourstruct;
     ourstruct.
}

/////////////////////////////////////////////////////////////////
// Pixel Sh
/////////////////////////////////////////////////////////////////
```

Member completion and red underlining



```
        //Find the color of the object
        float4 Color = tex2Dproj(ProjTexSampler, IN.TexCoord2 +

        //Give the textue a slightly blue tint
        Color.xy -= 0.1f;

        VS (
```

VS (**in A2V IN**, out V2P OUT) void
in A2V IN:
in A2V IN

```
    }
```

Parameter info