

Developer

/ Documentation

/ Unreal Engine ▾

/ Unreal Engine 5.4 Documentation

/ Designing Visuals, Rendering, and Graphics

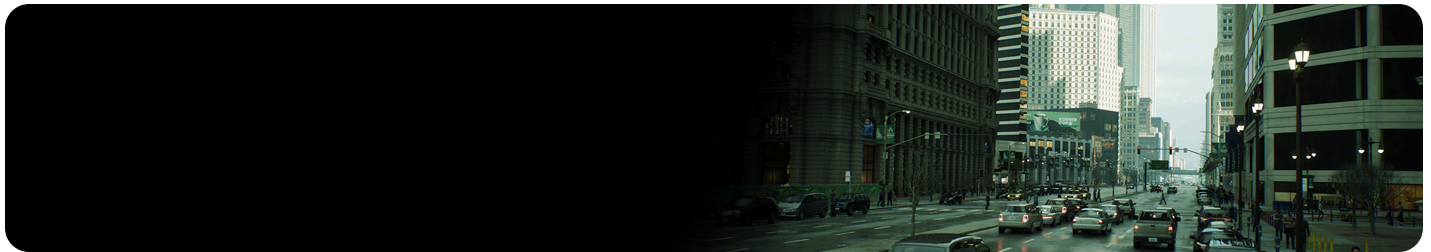
/ Graphics Programming

/ Shader Development

/ Cross Compiler

Cross Compiler

Information on the HLSLCC tool used to convert HLSL into GLSL.



This library compiles **High Level Shading Language (HLSL)** shader source code into a high-level intermediate representation, performs device-independent optimizations, and produces **OpenGL Shading Language (GLSL)** compatible source code. The library is largely based on the GLSL compiler from Mesa. The frontend has been heavily rewritten to parse HLSL and generate Mesa IR from the HLSL **Abstract Syntax Tree (AST)**. The library leverages Mesa's IR optimization to simplify the code and finally generates GLSL source code from the Mesa IR. The GLSL generation is based on the work in glsl-optimizer.

In addition to producing GLSL code, the compiler packs global uniforms in to an array for easy and efficient setting, provides a reflection mechanism to inform high level code which uniforms are required, and provides mapping information so that high level code may bind resources by index rather than by name at runtime.



UnrealBuildTool does not detect changes to external libraries, such as the HLSLCC. When you rebuild the HLSLCC library, add a space to OpenGLShaders.cpp to force the module to re-link.

The main library entry point is `HLSCrossCompile`. This function performs all steps needed to generate GLSL code from the source HLSL with the requested options. A summary of each stage follows:

Operation	Description
Preprocessing	The code is run through a C-like preprocessor. This stage is optional and may be omitted by using the <code>NoPreprocess</code> flag. Unreal performs preprocessing using MCPP before compilation and therefore skips this step.
Parsing	The HLSL source is parsed in to an abstract syntax tree. This is done in the function <code>_mesa_hlsl_parse</code> . The lexer and parser are generated by flex and bison respectively. See the section on parsing for more information.
Compilation	The AST is compiled in to Mesa intermediate representation. This process happens in the function <code>_mesa_ast_to_hir</code> . During this stage, the compiler performs functions such as implicit conversions, function overload resolution, generating instructions for intrinsics, and so on. A GLSL main entry point is generated. See <code>GenerateGlsIMain</code> . This stage will add global declarations for input and output variables to the IR, compute the inputs for the HLSL entry point, call the HLSL entry point, and write outputs to the global output variables.
Optimization	Several optimization passes are performed on the IR including function inlining, dead code elimination, constant propagation, elimination of common subexpressions, and so on. See <code>OptimizeIR</code> and especially <code>do_optimization_pass</code> for details.
Uniform packing	Global uniforms are packed in to arrays with mapping information retained so the engine may bind parameters to the relevant portion of the uniform array. See <code>PackUniforms</code> for details.
Final optimization	After uniforms have been packed, a second round of optimizations is run on the IR to simplify the code generated when packing uniforms.
Generate GLSL	Finally the optimized IR is converted to GLSL source code. The conversion from IR to GLSL is relatively straight-forward. In addition to

Operation	Description
	<p>producing definitions of all structs and uniform buffers and the source itself, a mapping table is written out in comments at the top of the file. This mapping table is parsed by Unreal to allow the binding of parameters. See <code>GenerateGlsI</code> and especially the <code>ir_gen_glsI_visitor</code> class for details.</p>

Parsing

The HLSL parser is built in two parts: the lexer and the parser. The lexer tokenizes the HLSL input by matching regular expressions to corresponding tokens. The source file is `hlsl_lexer.ll` and is processed by flex to produce C code. Each line begins with a regular expression followed by a statement written in C code. When the regular expression is matched, the corresponding C code is executed. State is stored in a number of global variables prefixed with "yy".

The parser matches rules to the tokenized input in order to interpret the grammar of the language and builds an AST. The source file is `hlsl_parser.yy` and is processed by bison to produce C code. Fully explaining the syntax used by bison is outside the scope of this document but looking at the HLSL parser should shed some light on the basics. In general, you define a rule as matching some sequence of tokens evaluated recursively. When a rule has been matched, some corresponding C code is executed allowing you to build your AST. The syntax within the C code block is as follows:

- `$$` = the result of parsing this rule, usually a node in the abstract syntax tree
- `$1`, `$2`, etc. = the outputs of the sub-rules matched by the current rule

When making changes to the lexer or parser, you must regenerate the C code using flex and bison. The `GenerateParsers` batch file handles this for you but you must setup the directories based on where flex and bison are installed on your system. The README file contains information on the versions I used and where binaries can be downloaded for Windows.

Compilation

During compilation, the AST is traversed and used to generate IR instructions. One important concept to grasp is that IR is a very low level sequence of operations. As such, it does not

perform implicit conversions or anything of that nature: everything must be done explicitly.

Some common functions of interest:

- **apply_type_conversion** - This function converts a value of one type to another if possible. Implicit versus explicit conversions are controlled via a parameter.
- **arithmetic_result_type**, et. al. - A set of functions that determine the result type of applying an operation to input values.
- **validate_assignment** - Determines if an rvalue can be assigned to an lvalue of a particular type. Allowed implicit conversions will be applied if necessary.
- **do_assignment** - Assigns an rvalue to an lvalue if possible using `validate_assignment`.
- **ast_expression::hir** - Converts an expression node in the AST to a set of IR instructions.
- **process_initializer** - Applies an initializer expression to a variable.
- **ast_struct_specifier::hir** - Builds an aggregate type to represent a declared structure.
- **ast_cbuffer_declaration::hir** - Builds a struct for the constant buffer layout and stores it as a uniform block.
- **process_mul** - Special code to handle the HLSL intrinsic `mul`.
- **match_function_by_name** - Looks up a function signature based on name and the list of input parameters.
- **rank_parameter_lists** - Compares two parameter lists and assigns a numerical rank indicating how closely the lists match. This is a helper function used to perform overload resolution: the signature with the lowest rank wins and a function call is declared ambiguous if any signature has the same rank as the lowest ranking signature. A rank of zero indicates an exact match.
- **gen_texture_op** - Handles method calls for builtin HLSL texture and sampler objects.
- **_mesa_glsl_initialize_functions** - Generates builtin functions for HLSL intrinsics. Most functions (e.g. `sin`, `cos`) generate IR code to perform the operation but some (e.g. `transpose`, `determinant`) leave in function calls deferring the operation to the driver's GLSL compiler.

Extending the Compiler

Here are some tips on implementing some types of features:

New Expressions

- Add an entry to the `ir_expression_operation` enum.

- In the `ir_expression` constructor handle your new expression to setup the typed result of the expression based on the types of input operands.
- If possible, add a handler to `ir_expression::constant_expression_value` to allow constant expressions to be evaluated at compile time.
- Add a handler to `ir_validate::visit_leave(ir_expression *ir)` to validate the correctness of the expression.
- Add an entry to the `GLSLExpressionTable` to map your expression to a GLSL expression.
- Modify the lexer to recognize the token(s) for your expression, if applicable.
- Modify the parser to recognize the token and create an appropriate `ast_expression` node, if applicable.

Intrinsics

- Add a builtin function definition to `_mesa_glsl_initialize_functions`.
- In most cases an intrinsic will map directly to a single expression. If that is the case, simply add a new `ir_expression` and use `make_intrinsic_genType` to generate the intrinsic function.

Types

- Add a `glsl_type` to represent your type within the IR. You can add this to `_mesa_glsl_initialize_types` or add it to one of the builtin type tables, e.g. `glsl_type::builtin_core_types`. For templated types see `glsl_type::get_sampler_instance` as an example.
- Modify the lexer to recognize the necessary token and the parser to match your token. See `Texture2DArray` as an example.
- Modify the parser to recognize the token and create the necessary type specifier. `texture_type_specifier_nonarray` is a good example.
- Modify `ast_type_specifier::hir` to perform any processing needed to create user-defined types. See the handling for structures as an example.
- Modify `ast_type_specifier::glsl_type` to return an appropriate `glsl_type`.
- If the type contains methods, modify `_mesa_ast_field_selection_to_hir` to handle them. See `gen_texture_op` as an example.


Attributes, flags, and qualifiers

- Add the attributes / flags / qualifiers to any IR and/or AST nodes where you will need them.
- Modify the lexer to recognize the necessary tokens.

- Modify the parser to add the grammatical rules as needed. E.g. if you were to add support for the [loop] attribute you'd modify the iteration_statement rule to accept an optional attribute preceeding it. Something like this: change iteration_statement to base_iteration_statement and add

iteration_statement:

```
1 iteration_attr base_iteration_statement
2 {
3   // result is the iteration statement
4   $$ = $2;
5   // apply attribute
6   $$->attr = $1;
7 }
8 base_iteration_statement
9 {
10  // pass thru if no attribute
11  $$ = $1;
12 }
```

 Copy full snippet

Finally, make modifications anywhere in the compiler where you need to know about the attribute.