|  |
| --- |
| Slic3r |
| Documentation  02/06/2016 |

1. Installation
2. Installing the Boost libraries

Download them from the Boost website. The official binaries won't work since they're compiled for MSVC. Extract the sources to **C:\dev\boost\_1\_60\_0 (**or whatever your version is). Enter that directory from the prompt and type:

**bootstrap.bat mingw**

**b2 toolset=gcc**

1. Installing Git for Windows

You can install Git by downloading this package:

Git for Windows

If asked, choose to run git from Windows prompt (adjust path environment) instead of git Bash. Line ending conversions don't matter, choose the default.

There are also several alternatives:

GitHub for Windows Simplified and GUI tools developed by GitHub team.

TortoiseGit GUI with shell integration. UI inspired by TortoiseSvn.

Msysgit. The port of the original Git library, provides command prompt and API. The GUI port listed above actually depends and includes a copy of msysgit.

1. Cloning the source

Open a command prompt and move to the directory where you want to create a Slic3r working copy. Then issue this command: git clone **git://github.com/alexrj/Slic3r.git** or git clone http://github.com/alexrj/Slic3r.git It will create a Slic3r directory in the current position.

1. Strawberry Perl

Download a copy from http://strawberryperl.com/ Select version 5.22.

Install it, following the on screen instruction. Basically that means keep clicking Next.

Now your command prompt is loaded with Perl. You start the command prompt with...

In the Start Menu, All Programs, then Accessories, finally Command Prompt

Press Win+R, type cmd, then Enter.

For Vista/Windows 7: open the Start menu, type cmd in the search box, then Enter.

Now continue with installing the dependencies section.

1. Citrus Perl

**Download CitrusPerl 5.14 (preferred over 5.12) from http://www.citrusperl.com/download.html**

Extract the zip to a folder. Choose a reasonable place, **like C:\dev\CitrusPerl**, as you won't be able to move it after the installation.

Run citrusutils.exe in **C:\dev\CitrusPerl\bin**

A window called citrusutils will be opened.

Click the "Set GCC" button to set the GCC compiler.

Select "Download the binary MinGW distribution used to build this Perl" and click Ok. It will now start downloading the compiler.

Select the Folder where you would like to install the compiler and click Ok. (You could ask it to extract to C:\dev\CitrusPerl too)

You can now close the citrusperl utilities.

Run citrusterm.bat in **C:\dev\CitrusPerl\bin** (it might be called citrusterm32.bat if you downloaded the 32-bit version), and you get a command prompt with Perl environment ready.

Then issue the following command:

**cpan JSON App::cpanminus**

You will need to open the citrusterm prompt every time you want to launch Slic3r, or run Perl for that matters. Now continue with the installing dependencies section.

1. Installing the dependencies

The Slic3r depends on few modules. We will be downloading, compiling and installing them. The Build.PL script will do the magic for us.

First, verify your Perl command environment is working. Start the prompt as stated above, and then type Perl -v. You should get some meaningful message. Check your Perl installation if not.

Now cd to the folder where Slic3r source code is cloned, and proceed with the build process to verify the dependence installed:

cd Slic3r

Perl Build.PL

Perl Build.PL --gui

1. Starting Slic3r

If you got the dependencies installed, now you could fire up the Slic3r with still in the command prompt by:

perl slic3r.pl

To start slic3r with a single click you should create a shortcut with a target that shows to the full path of the "wperl.exe" file and then the full path to Your "slicer.pl" file. For example, C:\dev\CitrusPerl\bin\wperl.exe C:\Users\YourUsername\Documents\GitHub\Slic3r\slic3r.pl

1. Install eclipse and epic Perl plugins

* Install jdk 8u94
* Install eclipse Tools for developers creating Java EE and Web applications Perl applications **https://eclipse.org/downloads/**
* Update eclipse
* Install epic Perl plugins on eclipse http**://www.epic-ide.org/download.php**
* Clone slic3r with eclipse git toolbox
* Download CDT eclipse (the CDT Project provides a fully functional C and C++ Integrated Development Environment based on the Eclipse platform. **https://eclipse.org/cdt/downloads.php**
* Open project and run project

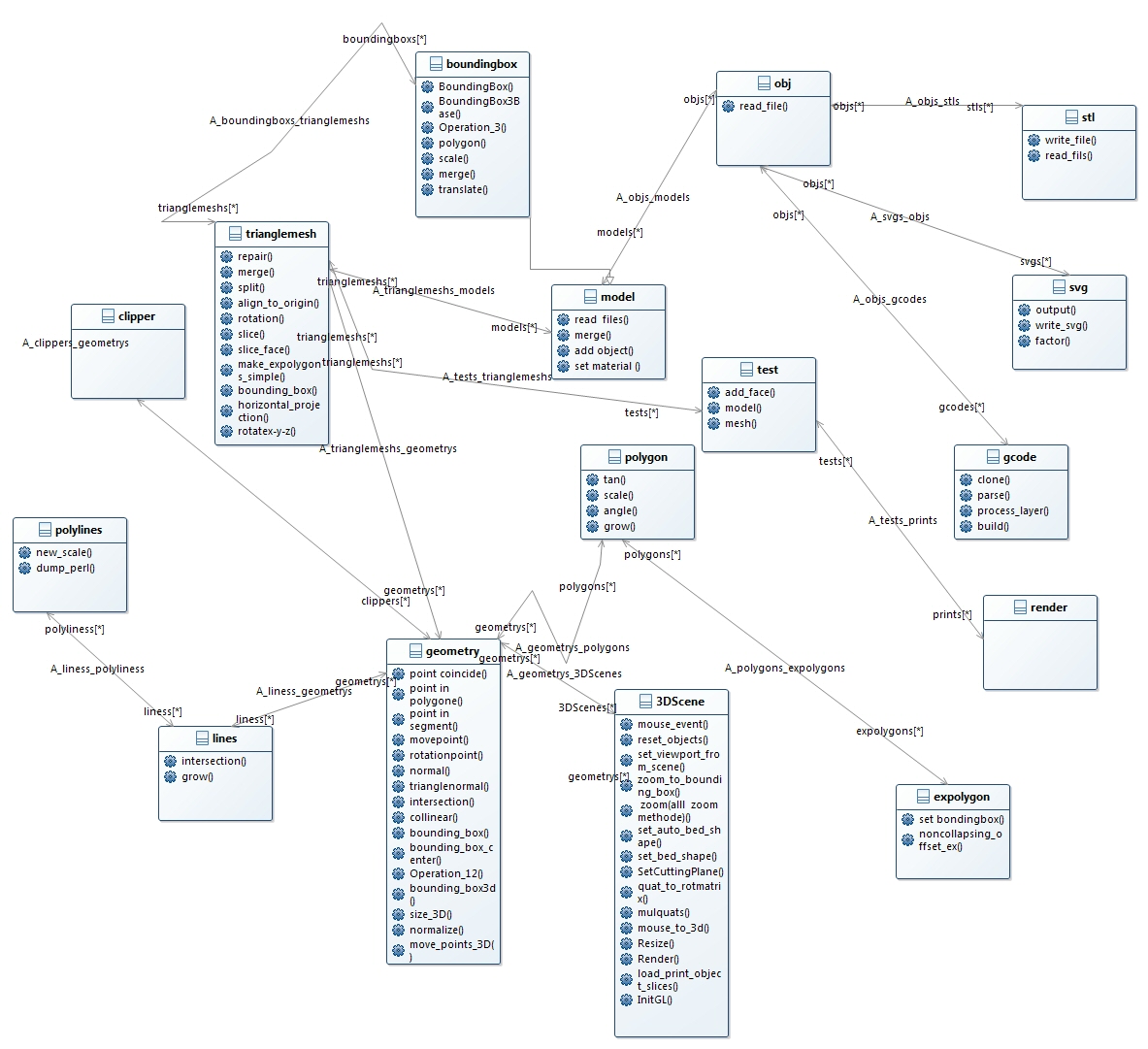
1. Class

Slic3r use Model view controller architecture and the application slic3r use the libraries clipper to calculate intersection and geometrical state.

The most important Modules are:

* Slic3r.pl the main class
* Lines, Polylines Polygon (in this this module the software manage all mesh of polygon, expolygon storage )
* Model (in this module the software manage all model in the view, we had riding file method, add object, merging model…)
* Bounding box ( )
* Geometry (manage all geometry value and coordinate in the view, angle, rotation,,, )
* Polylines collection
* Print (print models)
* SVG (manage the SVG generator)
* STL (manage the STL input files)
* Test (this module serve to test the model after modeling)

(Install and open programs)



1. Functioning
2. Perl XS

XS is an interface description file format used to create an extension interface between Perl and C code (or a C library) which one wishes to use with Perl. The XS interface is combined with the library to create a new library, which can then be either dynamically loaded or statically linked into Perl. The XS interface description is written in the XS language and is the core component of the Perl extension interface.

Link = <http://perldoc.perl.org/perlxs.html>

Link= <http://search.cpan.org/~dapm/perl-5.14.4/pod/perlxs.pod>

1. Using XS With C++

If an XSUB name contains **::** , it is considered to be a C++ method. The generated Perl function will assume that its first argument is an object pointer. The object pointer will be stored in a variable called THIS. The object should have been created by C++ with the new () function and should be blessed by Perl with the **sv\_set ref\_pv()** macro. A type map can handle the blessing of the object by Perl. An example type map is shown at the end of this section.

If the return type of the XSUB includes static, the method is considered to be a static method.

* It will call the C++ function using the **class::method() syntax.**
* If the method is not static the function will be called using the

**THIS->method() syntax.**

[**https://en.wikipedia.org/wiki/XS\_%28Perl%29**](https://en.wikipedia.org/wiki/XS_%28Perl%29)

1. **Testing module**

Testing is an important step in developing any important body of work. In today's pragmatic culture, we're taught to test first, test often, and design with tests. The expectation is that chanting "test " forgives all sins. To a large extent, this is true. Testing helps us produce quality software at all scales.

The extreme code produced by this extreme lifestyle hides in the test suite itself. Often the ugliest code we write resides in files with a .t extension. Riddled with redundant, ghastly expressions, the test suite is the collateral damage on our road to beautiful production code.

Let's review some common pitfalls made when testing. Many of these testing procedures may be new to you. Serious headway has been made in recent history with the testing libraries on the CPAN.

A Test File is Just a Program

Each test file is a program; just as important as any other program you'd write that uses software being tested. It must be treated with the same care. If you plan to use strict and warnings in a program related to the code you're testing, be sure to do the same in your tests.

Example

Let's assume you have just started to write a new module in Perl, and you would like to go with the flow and start by writing some automated test.

In order to make it simple I use the good old math example.

It is probably a good idea for any project to follow the best practices of the Perl community, and create the layout of the project as most of the CPAN modules are structured. This means you will have all the files of the project, the modules, the tests and all other auxiliary files in a single directory structure.

Within that structure the modules are going to be in the lib/ subdirectory, and the test scripts will be in the t/ subdirectory. The tests are simple Perl scripts but with a .t extension.

root/

lib/Math.pm

t/01\_compute.t

We created the first version of ours Math.pm module, and placed it in lib/Math.pm

**The module**

The lib/Math.pm file looks like this:

1. package Math;
2. use strict;
3. use warnings;
4. use 5.010;
6. use base 'Exporter';
7. our @EXPORT\_OK = qw(compute);
9. sub compute {
10. my ($operator, $x, $y) = @\_;
12. if ($operator eq '+') {
13. return $x + $y;
14. } elsif ($operator eq '-') {
15. return $x - $y;
16. } elsif ($operator eq '\*') {
17. return $x - $y;
18. }
19. }
21. 1;

I won ‘t go into détails now but You Can use It like This:

1. use 5.010;
2. use Math qw(compute);
3. say compute('+', 2, 3); # will print 5

**The Test**

The test script needs to call the compute() function and check if the resulting value is the same as we expected. To follow the conventions we create a file in the t/ directory with a .t extension:

**t/01\_compute.t** contrains the following:

1. use strict;
2. use warnings;
3. use 5.010;
5. use Test::Simple tests => 2;
7. use Math qw(compute);
9. ok ( compute('+', 2, 3) == 5 );
10. ok ( compute('-', 5, 2) == 3 );

Sources: <http://perlmaven.com/testing-a-simple-perl-module>

In the application every module had a script testing in packages Slic3r / t.

All modules were built in Slic3r/lib (libraries).