Slic3r
Il Perl e la stampa 3D
Perl & 3D printing
http://slic3r.org/
http://makerblog.it/
Alessandro Ranellucci
AAR@cpan.org
3D printing
Support material
Hello, I’m a 3D printer.

- Extruder Motor
- Extruder Heater
- XYZ Motors
- Heated Bed
Hello, I’m a 3D printer.

Electronics send electric signals to motors and heaters.
Hello, I’m a 3D printer.

Electronics sends electric signals to motors and heaters.

Computer transforms 3D model into instructions for printer.
Hello, I’m a 3D printer.

Computer transforms 3D model into instructions for printer

Human designs model and feeds it to the processing software

Electronics sends electric signals to motors and heaters
The toolchain

Internet → 3D model → CAM software (the so-called “slicer”) → G-code file → Firmware (inside printer electronics) → Motors & heaters

...hey, that’s Slic3r!
3D model formats

• **STL**: represents a mesh

• **OBJ**: represents surface colors too

• **AMF**: represents materials, plate positions etc.

• ...
The “slicer”
The “slicer”

• It takes a 3D model
• It cuts it into slices (layers)
• It generates the toolpaths
• It calculates the amount of plastic to extrude for each segment
M104 S200 ; set temperature
G28 ; home all axes
M109 S200 ; wait for temperature to be reached
G90 ; use absolute coordinates
G21 ; set units to millimeters
G92 E0
G1 Z0.400 F7800.000
G1 X85.700 Y85.880
G1 F1800.000 E1.00000
G1 X86.070 Y85.520 F540.000 E1.01518
G1 X86.270 Y85.350 E1.02290
G1 X86.690 Y85.030 E1.03843
G1 X87.370 Y84.620 E1.06177
G1 X87.840 Y84.400 E1.07703
G-code preview
- 12,000 lines of code
- 25,000 downloads for each release
- 90 forks on GitHub
- 746 issues (open and closed) on GitHub :-(
- 20 mails/day :-((((
- 80 users in #slic3r on FreeNode
- 1,354 posts in the forums
Is it possible...

• ...to handle computational geometry problems in Perl efficiently?

• ...to write desktop applications in Perl without compromises about the user experience?

• ...distribute multiplatform desktop applications in Perl?
Is it possible...

Is it convenient?

• ...to handle computational geometry problems in Perl efficiently?

• ...to write desktop applications in Perl without compromises about the user experience?

• ...distribute multiplatform desktop applications in Perl?
(Some) geometric algorithms used in Slic3r

- point in polygon
- point in segment
- nearest point
- points visibility in non-convex polygon
- line intersection
- merge collinear lines
- bounding box
- Douglas-Peucker simplification
- polygon packing
- union/difference/intersection/xor on polygons
- offset
- Voronoi diagram
- medial axis
- convex hull
- ...


Example: rectilinear infill generation

1. rotate polygon
2. compute bounding box
3. generate vertical lines
4. clip lines with the polygon contour
5. run a TSP search to connect clipped segments
Geometrical entities to represent

3D:
- Mesh
- Point
- Facet

2D:
- Point
- Line
- Polyline
  - ExtrusionPath
- Polygon
  - ExtrusionLoop
  - ExPolygon
- Surface
  - Region
  - Layer
Class instantiation frequency

When processing a simple cube, containing 8 vertices and 12 triangular facets:

Slic3r::Point is instantiated \(>2.000\) times
Slic3r::Polygon is instantiated \(1.850\) times

When processing a complex mesh, containing 298.370 vertices and 596.736 triangular facets:

Slic3r::Point is instantiated \(1.192.823\) times
Slic3r::Polygon is instantiated \(171.116\) times
The initial approach: Moose

```
package Slic3r::Point;
use Moose;
has 'x' => (is => 'rw');
has 'y' => (is => 'rw');

package Slic3r::Polygon;
use Moose;
has 'points' => (
    is => 'rw',
    isa => 'ArrayRef[Slic3r::Point]',
);
```

A simple cube required about 4 seconds to be processed (only measuring runtime).
package Slic3r::Point;
use Moo;

has 'x' => (is => 'rw');
has 'y' => (is => 'rw');

package Slic3r::Polygon;
use Moo;

has 'points' => (
    is => 'rw',
    default => sub { [] },
);

For a simple cube, processing time changed from 4 seconds (Moose) to about 0.8 with Moo.
The first optimizations: numerical constants

```perl
package Slic3r::Surface;
use Moo;

# top/bottom/internal
has 'type' => (is => 'rw');

package Slic3r::Surface;
use Moo;

use constant STYPE_TOP      => 0;
use constant STYPE_BOTTOM   => 1;
use constant STYPE_INTERNAL => 2;
has 'type' => (is => 'rw');

require Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(STYPE_TOP STYPE_BOTTOM STYPE_INTERNAL);
our %EXPORT_TAGS = (types => \@EXPORT_OK);
```
OOP DIY: blessed arrayrefs

```perl
package Slic3r::Point;

sub new {
    my $class = shift;
    my $self = [@_];
    bless $self, $class;
    return $self;
}

package main;

printf "x = %d, y = %d\n", @$point;

package Slic3r::Polygon;

sub new {
    my $class = shift;
    my $self = [@_];
    bless $self, $class;
    return $self;
}

package main;

printf "x = %d, y = %d\n", @$_ for @$polygon;
```

- Clean, readable syntax
- Lazy bless:
  ```perl
  my $point = algorithm_returning_an_arrayref();
bless $point, 'Slic3r::Point';  # only when I need OOP methods
  ```
my @points = map @$_,
    map @$_,
    grep $_[0]->encloses_point($point),
    grep $_[0]->area >= $threshold,
    @expolygons;
package Slic3r::Surface;

use constant S_EXPOLYGON    => 0;
use constant S_SURFACE_TYPE => 1;
use constant S_DEPTH_LAYERS => 2;

sub new {
    my $class = shift;
    my %args = @_
    my $self = [
        map delete $args{$_}, qw(expolygon surface_type depth_layers)
    ];
    $self->[S_DEPTH_LAYERS] // 1;
    bless $self, $class;
    $self;
}

# accessors
sub expolygon  { $_[0][S_EXPOLYGON]  }
sub surface_type { $_[0][S_SURFACE_TYPE] = $_[1] if defined $_[1];
                    $_[0][S_SURFACE_TYPE]  }

# delegate handles
sub encloses_point  { $_[0]->expolygon->encloses_point(@_)  }

For more complex objects: indexed arrayrefs

MooX::ArrayRef
Fighting memory waste

Devel::Size shows that **most of memory is used to store geometrical data** which was generated during the various processing steps, waiting to be used later.

Is there a way to “park” such data more efficiently?

```perl
my $packed_object = \pack 'cfa*'
  map $object->$_,
  qw(foo bar baz);

bless $packed_object, 'MyClass::Packed';
```
Packed classes

```perl
package Slic3r::ExtrusionPath;
use Moo;

has ...

use constant PACK_FMT => 'cfc';

sub pack {
    my $self = shift;
    my $o = \ CORE::pack PACK_FMT, map $self->$_, qw(foo bar baz);
    bless $o, 'Slic3r::ExtrusionPath::Packed';
    return $o;
}

package Slic3r::ExtrusionPath::Packed;

sub unpack {
    my $self = shift;
    my @data = CORE::unpack Slic3r::ExtrusionPath::PACK_FMT, $$self;
    return Slic3r::ExtrusionPath->new(
        map { $_ => shift @data } qw(foo bar baz)
    );
}
```
Summarizing: OOP models used in Slic3r

- **blessed arrayref** for intensively used geometrical entities

- **blessed packed scalars** for all geometrical entities which are stored massively but read infrequently

- **Moo classes** for high-level application logic and GUI
More techniques to try

**PDL**

Pros:
- mature, fast, efficient

Cons:
- need to reimplement algorithms too or live with overhead caused by conversions

**temporary files**

Serialize data structures to disk to avoid keeping them in memory. I/O overhead? MMap? This could allow to process models of any size without being limited by system memory.
Usage of C/C++ libraries

1. Prototyping in pure Perl

2. Porting to C o C++ via XS

2b. Using existing libraries via XS
Usage of C/C++ libraries

1. Prototyping in pure Perl

2. Identification of bottlenecks in terms of performance (Devel::NYTProf!) and memory (Devel::Size!); replacement with XS code

3. When multiple parts are ported to XS, overhead might be significant, so it could be convenient to port the entire algorithms (one abstraction level above)
Parallelization

Non-blocking single-threaded? NO.

- there's nothing blocking here!

- monolithic algorithms, too much overhead to return control to the event loop (we also have a GUI...)

- does not benefit from multi-core systems
Parallelization

Fork (worker pool?)

- overhead caused by IPC
- uncommon approach for GUI applications
Parallelization

Threads :-(

- Thread::Queue
  - worker pool model (task must be distributed at the beginning)

- efficient parallelization
  - how many threads?
  - only convenient above a certain number of cycles/items
  - Wx provides a powerful API for GUI threading

- problems:
  - heavy usage of memory (not released to the system)
  - need to duplicate logic to support non-threaded perls too

Idea for the future: handle threading in C code directly?
GUI & user-experience

- goal: double interface (GUI + CLI)

- keep GUI optional: $no_gui if !eval "use Wx; 1"

- wxPerl, the most native framework on all platforms

- need to write OOP wrappers around Wx widgets (Moo!)
Packaging and distribution: goals

1. pre-compiled executables, ready to be launched with a simple “double click” and no dependencies to install
   - ...on MacOS X, Windows, Linux
   - ...on both 32bit and 64bit!

2. ease of installation and execution of plain Perl code for the “early testers” grabbing source files form GitHub
Cava Packager

- creates self-contained executable binaries (they must be compiled on each platform)

- works flawlessly with Wx code and XS modules

- creates native installers

- embeds icons and support files

- soon released under GPL
Installing dependencies from CPAN (for the masses)...

• :-(

• painful misconfigured CPAN clients on many operating systems/distros

• users are non Perl-savvy, and they have often a very negative attitude about it

• they’re not patient enough to fix their CPAN clients or use perlbrew: the usual answers provided by/inside the mongers community are not applicable in the outer world
Installing dependencies from CPAN (for the masses)...

• the best compromise...
  curl -kL http://install.perlbrew.pl | bash
cpanm < deps.txt

• ideas for the future: automatic cross-compilation setup (cloud?)

• ideas for the future: Ubuntu/Debian packages
Conclusions

• Perl is an excellent language for developing multiplatform desktop applications

• ...but nobody knows

• The negative attitude about Perl should be actively fighted with excellent and cool usage cases and not just ignored

• Perl is an excellent language for prototyping complex algorithms that can be optimized using XS just where needed
¿ Q&A ?