Skydiving simulation with Perl

FREE FALL

Computer game programmers apply physical formulas and special tricks to create realistic animations. Simple DirectMedia Layer (SDL), which is available as a Perl wrapper, provides a powerful framework for creating simple 2D worlds with just a couple of lines of code [1]. **BY MICHAEL SCHILLI**

After ripping an old VHS cassette of my tandem skydive, posting it on YouTube, and mailing the link [2] to a couple of guys at work, a debate about the physical laws that apply during a parachute jump ensued.

In a simplified model that ignores crosswinds, the jumper starts with a vertical speed of $v_y = 0$ and immediately starts to accelerate because of gravity. Drag, which grows proportionally with the skydiver's downward speed, counteracts gravity [3]. Depending on the skydiver's weight and proportions, a balance of forces occurs at around 180 km/h, and the downward speed becomes constant. At this point, skydivers feel like they are floating in space, and this state continues until the chute opens, which feels like they are being pulled upward with a rope.

Free Fall

The skydive script in Listing 1 [4] simulates a parachute jump. An icon *dive.png* represents a jumper in free fall. Jumpers start off slowly and accelerate until they reach a constant terminal speed (v_{term}) of 50 m/s (180 km/h). Users can press the up arrow key to open the chute. At this point, the icon changes into *para.png*, a skydiver with an open chute. The diver decelerates quickly at first and then slowly floats down to the ground (Figures 1-3).

The script counts down the seconds from jumping to safe landing. The idea is to pull the ripcord as late as possible but to make sure the impact speed is less than 3 m/s (about 11 km/h) to avoid injury to the skydiver. The display shows the elapsed time in seconds on the left, and the current drop speed in meters per second on the right. Record times display below the current counter and they stay there until a new attempt beats them. If the player is too fast, the chute icon turns back into a skydiver without a chute on the ground to indicate an invalid attempt. Of course, a failed attempt will not change the high-score time (Figures 4 and 5).

The Physics of Free Fall

The speed of a body accelerating from a standstill is $v = a \cdot t$. In the case of a body jumping out of a plane, the acceleration a is equal to gravity (9.82 m/s^2); the time t is counted down in seconds and starts with the jump. The aerodynamic drag counteracting gravity can be described as a negative acceleration, which is zero for $v_y = 0$ and equivalent to gravity (9.81 m/s^2) for $v_y = v_{\text{term}}$.

Aerodynamic drag is calculated with reference to mass, speed, and the skydiver's coefficient of friction in the air. According to "The Free Fall Research Page" [5], an adult weighing 80 kilograms accelerates to about 190 km/min within 14 seconds after jumping and covers a distance of 548 meters in that time. After this, the skydiver falls at a constant speed of 3000 m/min until the chute opens.

However, the terminal velocity depends on the flight position. Drag is lower if the skydiver jumps head down; speeds of more than 200 km/h are easily achieved. More details on calculating drag are available online [3]. In this situation, the body is moving at constant speed because drag counteracts gravitational force. Now the skydiver covers a

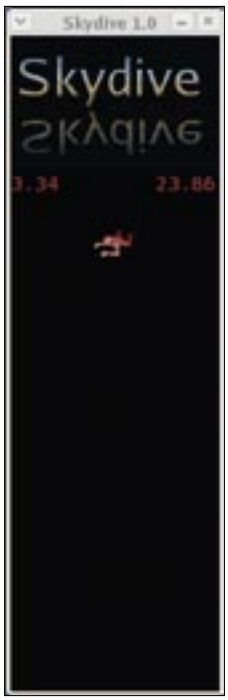

Figure 1: The skydiver accelerates after the jump and reaches constant speed after a couple of seconds.

distance of $s = v \cdot t$ in time t at speed v.

The Physics of the Game

In an animated game that draws 50 frames per second, you do not need to multiply to calculate a smooth trajectory for the figure between two frames. Simply dividing the speed in meters per second by 50 and adding the result to the current position gives you the new position. If you repeat this 50 times for every frame, the result at the end of one second will be exactly $s = v \cdot 1s$, which matches the physical formula for constant motion.

Because of the relatively short gaps between the individual frames, this calculation even works for the uniformly accelerated motion of a body falling to earth. Of course, it is not only the position of the body that changes in every frame, but the speed, too. To allow for this change, I simply add 1/50th of gravity to the current speed for every frame. Repeating this 50 times gives me exactly $v = a \cdot 1s$.

Drag and Drop

To make things worse, acceleration isn't constant. If the body drops out of a plane, acceleration is 9.81 m/s^2 if you ignore effects such as crosswinds and lift. The greater the speed, the greater the effect of drag on the drop and the lower effective downward acceleration will be. Once the body has reached a terminal velocity of v_{term}, acceleration drops to zero and the skydiver falls at constant speed. The game solves this problem by applying a simplified method. The *deceleration()* function calculates a value that is then subtracted from the current acceleration. The value is calculated with reference to the current and maximum speeds as a linear relationship.

When the chute opens, acceleration becomes negative. However, the chute can't apply an arbitrary braking force. The game limits the maximum counterforce to 2*g*. Of course, what *deceleration()* does isn't exactly accurate, but it is fine for the game.

Figure 2: The jumper is falling at a speed of 40.96 m/s and isn't far from the ground.

Blit the Image

When an icon moves through the playing field – like the skydiver dropping out of the sky, for example – SDL first deletes the old entry and then redraws the image at the new position. The icon is stored in memory as an image, and the *blit()* method just copies it from one memory position to another. This trick means that changes on the gaming screen can occur at an impressive speed, and the user has the illusion of a real world.

The game logo at the top of the playing area is a PNG graphic I created with GIMP. The script loads the *logo.png* file from disk into memory in line 33 with the *SDL::Surface* class constructor. Line 59 defines an *SDL::Rect* class rectangle including length, width, and the position of the graphic on screen. *X* coordinates run from left to right and *Y* coordinates from the top down. The *blit()* method for the graphic in *$logo* in line 65 copies the data to the playing area, *$app*.

SDL doesn't refresh immediately, though. For performance reasons,

Figure 3: The chute opens and slows the fall. The skydiver's impact speed should be less than 3.0 m/s for a safe landing.

SDL waits until the programmer tells it to refresh by calling *update()*. This means that SDL can refresh many rectangles at the same time, giving the viewer the impression of a smooth animation.

Main Loop of Life

Line 7 sets the speed of the animation to 20 milliseconds per frame, which is equivalent to 50 frames per second, as reflected in the *$FRAMES_PSEC* variable in line 8. The infinite loop starting in line 93 displays the frames on screen. To keep time, the script uses *$app->ticks()* to query the number of milliseconds that have elapsed since the program started and stores the result in the *$synchro_ticks* variable.

Another measurement at the end of the loop determines how many milliseconds have elapsed between the start and the end of the loop. If the number is less than 20, the script has to wait until the allowance of 20 milliseconds per frame has elapsed. To insert gaps on a millisecond scale so that the animation runs smoothly, you can use *select()*. If the difference between the allotted time and the elapsed time is negative, the calculations inside the loop have taken longer than 20 milliseconds and you need to rewrite the script or reduce the frame rate.

While the skydive program is busy with the main loop, events such as key presses, mouse moves, or clicks on the window close button are passed in to the application. The *SDL::Event* object defined in line 69 provides the *poll()* method, which tells me whether an event is waiting. *event_type()* gives me the event type, for example, *SDL_QUIT*, which occurs if the user clicks to close the application window. In this case, the script simply terminates with a call to *exit* in line 137.

Type *SDL_KEYDOWN* events indicate that the user has pressed a key. *key_name* in line 142 discovers which key it was. Fortunately, SDL translates key

codes to handy strings, returning a value of *right* when the right arrow key was pressed, and *q* if somebody hit the *q* key.

The *set_key_repeat()* method helps handle longer key presses as repeat input and expects two parameters.

The first parameter specifies how long a key must be held down to be evaluated by SDL as continuous fire.

Listing 1: skydive

```
001 #!/usr/bin/perl -w
002 use strict;
003 use SDL;
004 use SDLMove;
005 use SDL::TTFont;
006
007 my $SPEED_MS    = 20;
008 my $FRAMES_PSEC =
009   1000.0 / $SPEED_MS;
010 my $VTERM_FREE =
011   50;    # Terminal speed
012 my $VTERM_PARA =
013   3;     # ... with parachute
014 my $WIDTH    = 158;
015 my $HEIGHT   = 500;
016 my $G        = 9.81;
017 my $MAX_LAND = 3.1;
018
019 my $bg_color =
020   SDL::Color->new(
021   -r => 0,
022   -g => 0,
023   -b => 0
024   );
025 my $fg_color =
026   SDL::Color->new(
027   -r => 0xff,
028   -g => 0x0,
029   -b => 0x0
030   );
031
032 my $logo =
033   SDL::Surface->new(
034   -name => "logo.png");
035
036 # Load player icons
037 my $diver =
038   SDL::Surface->new(
039   -name => "dive.png");
040 my $para =
041   SDL::Surface->new(
042   -name => "para.png");
043
044 my $app = SDL::App->new(
045   -title  => "Skydive 1.0",
046   -depth  => 16,
047   -width  => $WIDTH,
048   -height => $HEIGHT
049 );
050
051 my $font = SDL::TTFont->new(
052   -name =>
053 "/usr/X11R6/lib/X11/fonts/
TTF/VeraMono.ttf",
054   -size => 15,
055   -bg   => $bg_color,
056   -fg   => $fg_color
057 );
058
059 my $lrect = SDL::Rect->new(
060   -width  => $logo->width,
061   -height => $logo->height,
062   -x      => 0,
063   -y      => 0
064 );
065 $logo->blit(0, $app, $lrect);
066 $app->update($lrect);
067
068 my $event =
069   new SDL::Event->new();
070 $event->set_key_repeat(200,
071   10);
072
073 my $record_time;
074 my $gtime;
075
076 # Next game ...
077 GAME: while (1) {
078
079   my $obj = SDLMove->new(
080     app      => $app,
081     bg_color => $bg_color,
082     x        => $WIDTH / 2 -
083       $diver->width() / 2,
084     y     => $logo->height,
085     image => $diver
086     ,    # Start with diver
087   );
088
089   my $v     = 0;
090   my $vterm = $VTERM_FREE;
091   my $start = $app->ticks();
092
093   while (1) {    # Frame loop
094     my $synchro_ticks =
095       $app->ticks;
096
097     # Accelerate
098     $v += (
099       $G - deceleration(
100       $v, $vterm
101       )
102     ) / $FRAMES_PSEC;
103
104     # Move player downwards
105     $obj->move("s",
106       $v / $FRAMES_PSEC);
107
108     if ($obj->hit_bottom()) {
109       if ($v <= $MAX_LAND)
110       {    # soft enough?
111         if (!defined $record_time
112           or $gtime <
113           $record_time)
114         {
115           $record_time = $gtime;
116         }
117         nput($app, 0,
118           $lrect->height + 20,
119           $record_time);
120       } else {
121         $obj->wipe();
122         $obj->image($diver);
123         $obj->move(
124           "s",    # indicate crash
125           $para->height -
126             $diver->height
127         );
128       }
129       sleep 5;
130       $obj->wipe();
131       next GAME;
132     }
133
134     # Process all queued events
135     while ($event->poll != 0) {
136       my $type = $event->type();
137       exit if $type == SDL_QUIT;
138
139       if ($type == SDL_KEYDOWN)
140       {
141         my $keypressed =
142           $event->key_name;
143
```

linux.conf.au
is coming home.

The University of Melbourne
28 January - 2 February 2008

http://linux.conf.au

The second parameter specifies the gap between rounds of fire, again in milliseconds. If you want to add a function to move the skydiver to the left or right, this ability would help.

When a user presses the up arrow key, this is the signal to open the chute, and the *elsif* condition in line 152 triggers two actions. The terminal velocity *$VTERM* is reduced from *$VTERM_FREE* to *$VTERM_PARA*. The *image()* method for the player object, *$obj*, sets the player icon to *$para*, the parachute icon.

Valid key presses include *r* for restart (abort current jump and perform a new one), and *q* for quit. The left and right arrow keys are defined to let players move the skydiver left or right, which can be used to extend the game.

The variable *$gtime* holds the time of the current round, and *$record_time* accepts a new value if the player achieves a new record time, but without too hard a landing. The application itself is represented by the *SDL::App* type object *$app*, a class derived from *SDL::Surface*. Drawing actions in the application window or refreshes of modified rectangles use the *$app* object.

To simplify the process of moving the player icon, the *SDLMove.pm* module in Listing 2 [4] defines a couple of auxiliary functions. The *image()* method draws the player icon on the specified *SDL::Surface* type object.

The SDLMove module knows the dimensions of the app, so it can provide *hit_bottom()* to tell whether the player has reached the bottom edge of the field, indicating the round is over. The *wipe()* method removes the player icon from the field in one fell swoop, for example, to change a failed skydiver into a free-falling icon at the bottom of the screen, showing that the player messed things up. The *move()* method moves the figure by the specified number of pixels in a specific direction (*n* = north, *s* = south, *w* = west, *e* = east). The parameters can

Figure 4: Here is a safe landing at 3.0 m/s and the player set a new record of 17.60 seconds.

Figure 5: Good time, but it doesn't count because the impact speed was 45.33 m/s – far too fast!

Listing 1: skydive

```
144     if (
145      $keypressed eq "left")
146     {
147      $obj->move("w", 0.1);
148     } elsif (
149      $keypressed eq "right")
150     {
151      $obj->move("e", 0.1);
152     } elsif (
153      $keypressed eq "up")
154     {
155
156      # deploy parachute
157      $vterm = $VTERM_PARA;
158      $obj->image($para);
159     } elsif (
160      $keypressed eq "r")
161     {
162      $obj->wipe();
163      next GAME;
164     } elsif (
165      $keypressed eq "q")
166     {
167      exit 0;    # quit
168     }
169    }
170   }
171   $gtime =
172     ($app->ticks - $start) /
173     1000.0;
174
175   nput($app, 0,
176    $lrect->height, $gtime);
177   nput($app, 110,
178    $lrect->height, $v);
179
180   my $wait = $SPEED_MS -
181     ($app->ticks -
182      $synchro_ticks);
183   select undef, undef, undef,
184     $wait / 1000.0
185     if $wait > 0;
186  }
187 }
188
189 ################################
190 sub deceleration {
191 ################################
192  my ($v, $vterm) = @_;
193
194  my $d = $v / $vterm * 9.81;
195
196  $d = 0      if $d < 0;
197  $d = 2 * $G if $d > 2 * $G;
198
199  return $d;
200 }
201
202 ################################
203 sub nput {
204 ################################
205  my ($app, $x, $y, $number) =
206    @_;
207
208  my $rect = SDL::Rect->new(
209   "-height" => $font->height,
210   "-width"  =>
211     $font->width($number),
212   "-x" => $x,
213   "-y" => $y
214  );
215
216  $app->fill($rect,
217   $bg_color);
218  my $string =
219    sprintf "%-5.2f", $number;
220  $font->print($app, $x, $y,
221   $string);
222  $app->sync();
223 }
```

contain pixel fractions that will not affect the current movement but will be accumulated by the script for future actions. Before moving the player icon, SDLMove deletes the old image to project a smooth movement onto the screen.

Configuration

Back in Listing 1, *$VTERM_FREE* specifies a terminal velocity in free fall of 50 m/s. *$VTERM_PARA* sets the drop rate of 3 m/s for the chute, which the chute will achieve after some time gliding. In the section following line 7, you can change these values and some other parameters, such as the height and width of the animation window.

To be able to display text on screen, the *SDL::TTFont* module juggles with True Type fonts; the module renders text strings and helps drop them on the playing field.

The constructor called in line 51 loads the fixed font VeraMono, which is stored in the *TTF* subdirectory below my X server's font directory. On Debian systems, the font path is different, and you will need to add */usr/share/fonts/truetype/ttf-bitstream-vera/VeraMono.ttf*. Also note that Debian systems come with a broken SDL Perl wrapper. The downloadable version of the script contains the necessary adjustments to compensate for this flaw.

The *-fg* and *-bg* options set the font color to red on a black background. The *print()* method handles rendering and displays the text at coordinates *$x, $y* on the playing field. Like the rectangles referred to previously, SDL does not refresh the display directly after a print command but waits for the programmer to *sync()* the *$app* object.

If one call overwrites the same position with new text, the original display is kept, and after a number of iterations, the numeric field is jumbled. The *nput* function defined in line 203 determines the size of the rendered text string and defines an enclosing rectangle, then paints the rectangle black to allow the *print()* function to write over it.

INFO

[1] SDL wrapper for Perl: *http://arstechnica.com/guides/tweaks/games-perl.ars*

[2] YouTube video showing the Perlmeister skydiving: *http://youtube.com/watch?v=aRxvsSs0sz4*

[3] Drag: *http://en.wikipedia.org/wiki/Drag_(physics)*

[4] Listings and Icons: *http://www.linux-magazine.com/Magazine/Downloads/85*

[5] "Free Fall – Falling Math": *http://www.greenharbor.com/fffolder/math.html*. Green Harbor Publications, 2005.

[6] Frozen Bubble: *http://www.frozen-bubble.org*

Installation

SDL is included with most popular Linux distributions; if not, you will need to install the *SDL*, *SDL-devel*, *SDL_ttf*, *SDL_ttf-devel*, and *SDL_mixer* packages. Then complete the install of the Perl wrapper and the necessary SDL modules by calling *install SDL_perl* in a CPAN shell.

Install all of the aforementioned libraries before installing SDL_Perl or you will

be missing True Type font support. The three icons – *logo.png*, *dive.png*, and *para.png* – are available online [4]. The script will look for the icons below the current directory when launched and complain if it can't find them.

Extensions

With just a couple of lines of Perl code, you could easily extend the game. If you are interested in more tips from experts, I suggest that you take a look at the Frozen Bubble game [6] source code. Frozen Bubble includes professional animations and was written with SDL_Perl.

To add more realism to the skydiving prototype, you could allow the skydiver to jump from a plane moving at a certain horizontal speed. In this case, the skydiver would move laterally at a constant speed with drag counteracting the movement. The aim of the game would be to achieve a soft landing and to hit a target on the ground or to avoid water or power lines.

The skydiver could slowly maneuver after opening the chute. Also, you could add a crosswind to make things more difficult and use SDL_mixer to generate sound effects. ■

Listing 2: SDLMove.pm

```
001 package SDLMove;
002 use strict;
003 use warnings;
004 use SDL;
005 use SDL::App;
006
007 ##################################
008 sub new {
009 ##################################
010  my ($class, %options) = @_;
011
012  my $self = {%options};
013  bless $self, $class;
014
015  $self->image(
016   $self->{image});
017  return $self;
018 }
019
020 ##################################
021 sub image {
022 ##################################
023  my ($self, $image) = @_;
024
025  $self->{image} = $image;
026  $self->{drect} =
027    SDL::Rect->new(
028   -width  => $image->width,
029   -height => $image->height,
030   -x      => $self->{x},
031   -y      => $self->{y},
032    );
033 }
034
035 ##################################
036 sub move {
037 ##################################
038  my ($self, $direction,
039   $pixels)
040    = @_;
041
042  my $rect = $self->{drect};
043  my $app  = $self->{app};
044
045  if ($direction eq "w")
046  {    # left
047   $self->{x} -= $pixels
048     if $self->{x} > 0;
049
050  } elsif ($direction eq "e")
051  {    # right
052   $self->{x} += $pixels
053     if $self->{x} <
054     $app->width -
055     $rect->width;
056
057  } elsif ($direction eq "n")
058  {    # up
059   $self->{y} -= $pixels
060     if $self->{y} > 0;
061
062  } elsif ($direction eq "s")
063  {    # down
064   $self->{y} += $pixels
065     if $self->{y} <
066     $app->height -
067     $rect->height;
068  }
069
070  $self->{old_rect} =
071    SDL::Rect->new(
072   -height => $rect->height,
073   -width  => $rect->width,
074   -x      => $rect->x,
075   -y      => $rect->y,
076    );
077
078  $rect->x($self->{x});
079  $rect->y($self->{y});
080  $app->fill(
081   $self->{old_rect},
082   $self->{bg_color}
083  );
084
085  $self->{image}
086    ->blit(0, $self->{app},
087   $rect);
088  $app->update(
089   $self->{old_rect}, $rect);
090 }
091
092 ##################################
093 sub wipe {
094 ##################################
095  my ($self) = @_;
096
097  $self->{app}->fill(
098   $self->{drect},
099   $self->{bg_color}
100  );
101  $self->{app}
102    ->update($self->{drect});
103 }
104
105 ##################################
106 sub hit_bottom {
107 ##################################
108  my ($self) = @_;
109
110  return $self->{y} >
111    $self->{app}->height -
112    $self->{drect}->height;
113 }
114
115 1;
```