High Art
on top of
LOW-LEVEL APIS

Building Games with Ruby

Andrea O. K. Wright, Chariot Solutions
www.chariotsolutions.com

I'm going to talk about building games with Ruby.

High Art

on top of

LOW-LEVEL APIS

Building Games with Ruby

Agenda

1. Buiding 2D Games with Ruby
2. Building 3D Games with Ruby
3. Is Ruby a legitimate player in the gaming space?

I'm going to start off by talking about libraries geared towards 2D game development, then we'll move on to 3D game development. I'll conclude by discussing whether playing games written in Ruby can be as much fun as writing them -- or do they just run too slowly with too many breaks in the action at inopportune times.

This is not a tutorial about how to use any particular library, but I do hope to give you an idea of what the different libraries have to offer and what distinguishes one from another.

Ruby/SDL

Creator and Lead Developer: Ohai

http://www.kmc.gr.jp/~ohai/rubysdl.en.html

The first library I'm going to talk about is Ruby/SDL, a Ruby extension for SDL by a developer who goes by Ohai.

SDL stands for Simple Directmedia Layer. It's is a cross-platform, open source library that provides multimedia support.

Like most of the libraries I'm going to talk about it offers sound and video support, but for the most part I'm not going to talk about those aspects of game development this afternoon.

SDL was developed in C. Making C library functions available to Ruby developers involves using the Ruby C API and wrapping the original C function in ways that account for the differences between the two languages.

Ruby/SDL C Extension


```
Uint32 SDL_GetTicks(void);
```

In some cases, the original C doesn't require much in the way of special handling to to be callable from the Ruby-side -- like sdl_getTicks, which returns how long the program has been running, in milliseconds. Here's that C method's signature frop the original SDL library.

Ruby/SDL C Extension


```
Uint32 SDL_GetTicks(void);
```

Ruby/SDL

```
static VALUE sdl_getTicks(VALUE mod)
{
  return UINT2NUM(SDL_GetTicks());
}
```

And here in the bottom box is the Ruby\SDL C Extension wrapper for sdl_getTicks. The UINT2NUM Macro is used to convert the C return type to a Ruby Num for consumption by the Ruby caller.

Also, notice that that even though the C function doesn't take any arguments, the corresponding wrapper takes one parameter. Not being object-oriented, C wouldn't know what to call the function on if it was not passed in.

The wrapper also needs to be referenced in the extension's init method in order to be callable from the Ruby side.

Ruby/SDL C Extension


```
Uint32 SDL_GetTicks(void);
```

Ruby/SDL

```
static VALUE sdl_getTicks(VALUE mod)
{
  return UINT2NUM(SDL_GetTicks());
}
```

```
rb_define_module_function(mSDL,
                  "getTicks", sdl_getTicks,0);
```

So here is how that wrapper is referenced in the ruby\sdl initialization code.

The rb_define_module_function maps the C extension code to a corresponding method name the Ruby programmer can use, in this case “getTicks”.

Ruby/SDL C Extension


```
Uint32 SDL_GetTicks(void);
```

Ruby/SDL

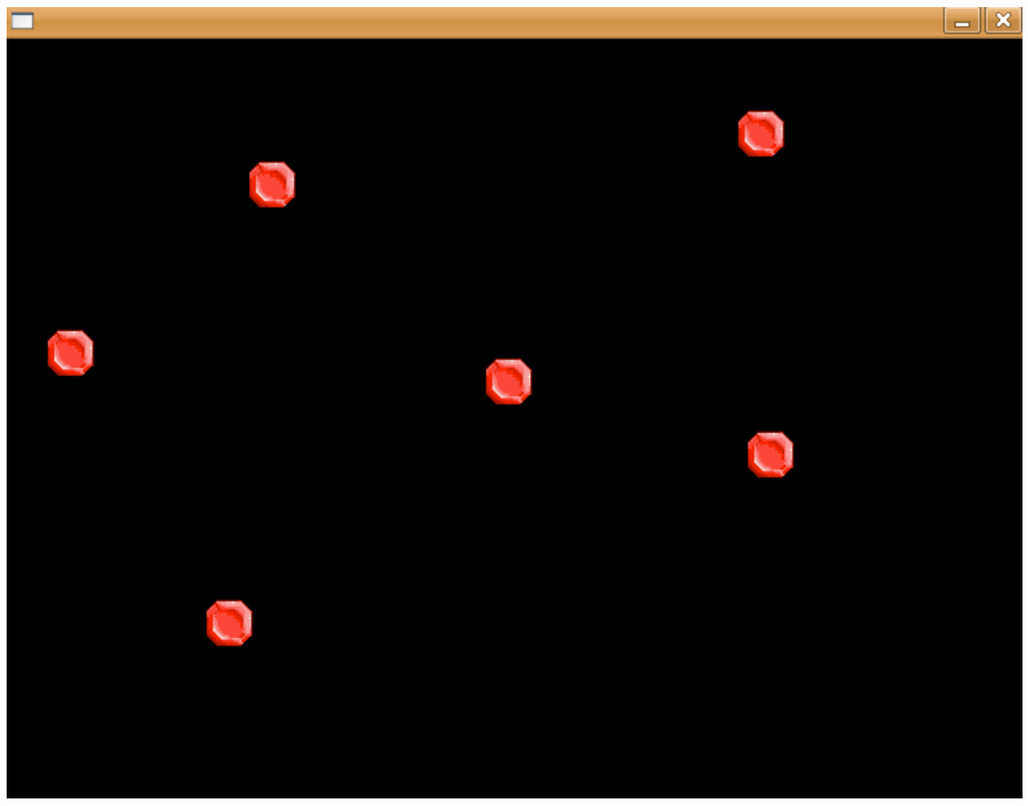
```
static VALUE sdl_getTicks(VALUE mod)
{
  return UINT2NUM(SDL_GetTicks());
}
```

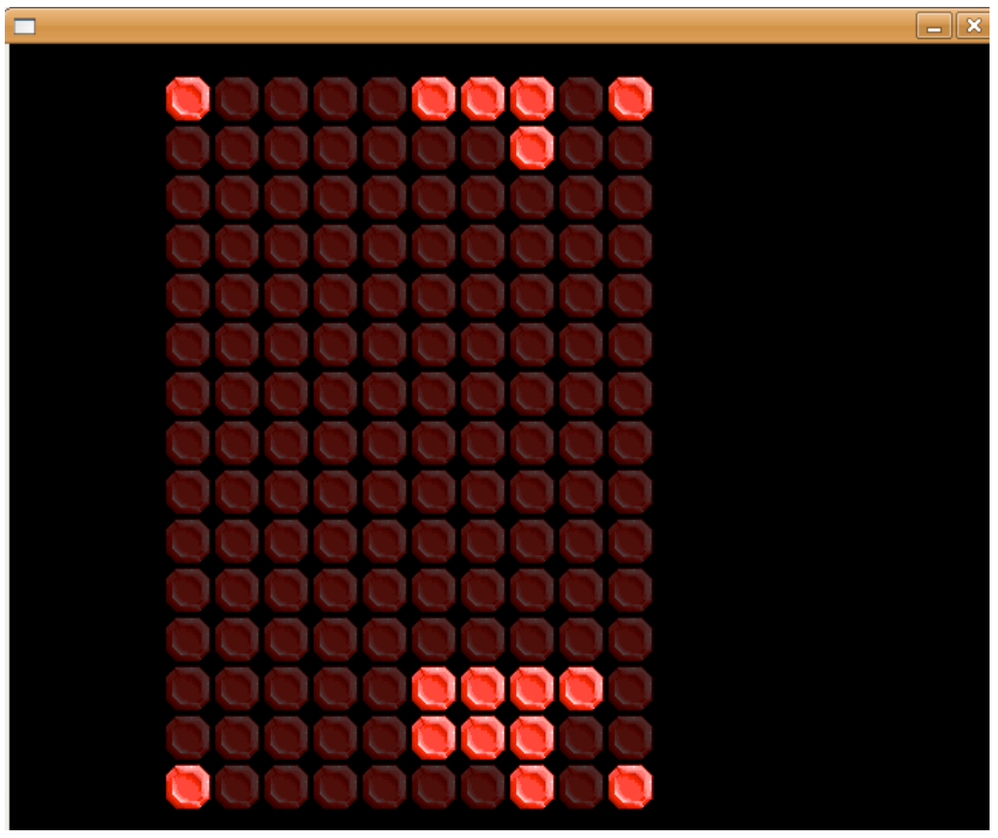
```
rb_define_module_function(mSDL,
                "getTicks", sdl_getTicks,0);
```

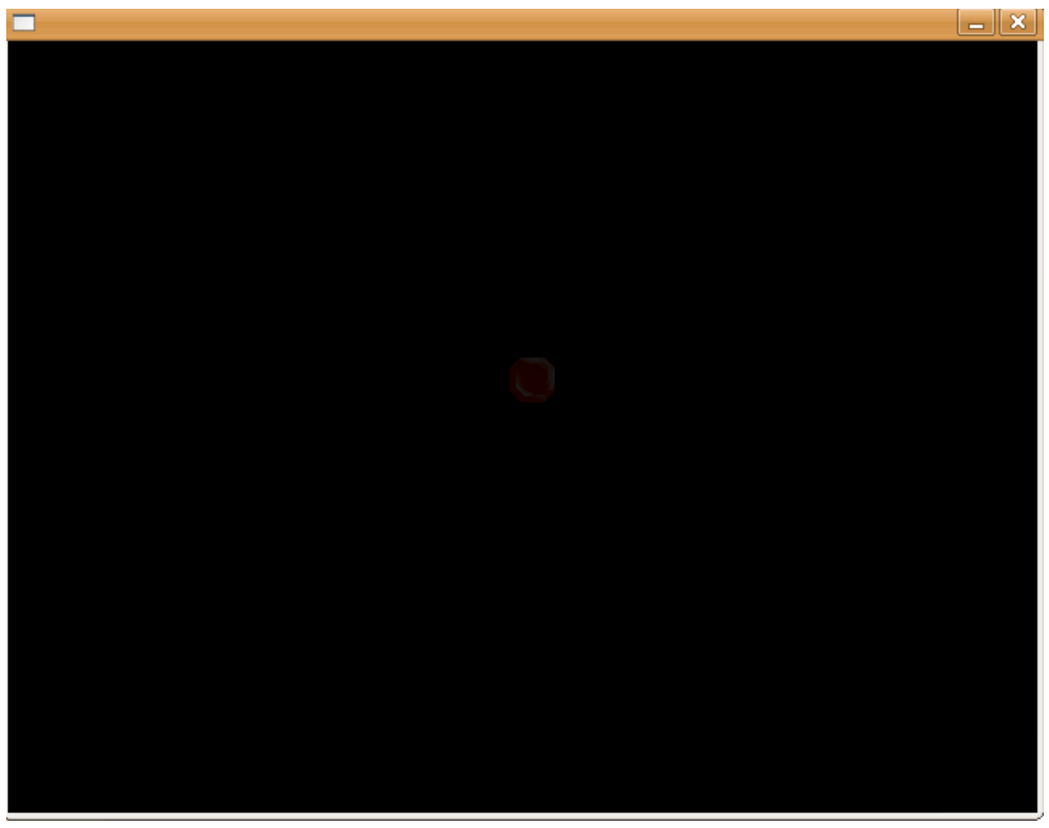
```
alias get_ticks getTicks
```

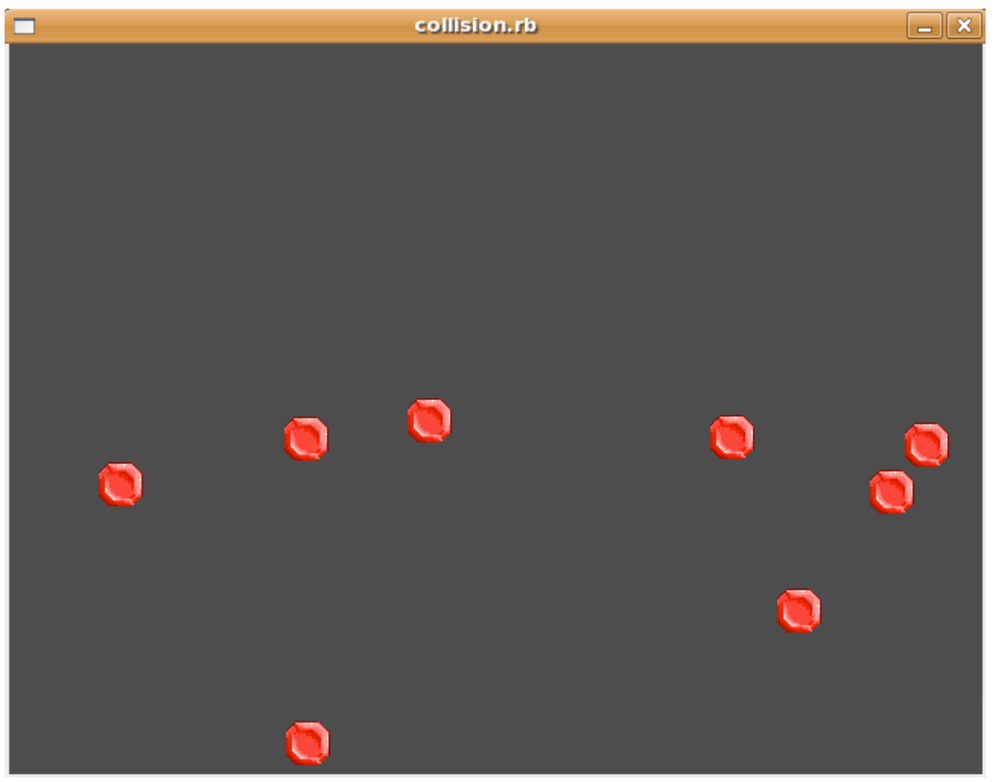
For people who prefer to use Ruby-style method names , with underscores, Ohai provides an alias, since you can’t pass multiple Ruby method names to rb_define_module_function.

Ruby/SDL Samples









Here are four of the sample apps that are packaged with SDL. They are pretty basic but they represent a good chunk of the functionality you would need to create a wide variety of more complex games.

The red shapes are set to move randomly except for the one in the middle -- which is wired to move up, down, left or right depending on which of the arrow keys you press.

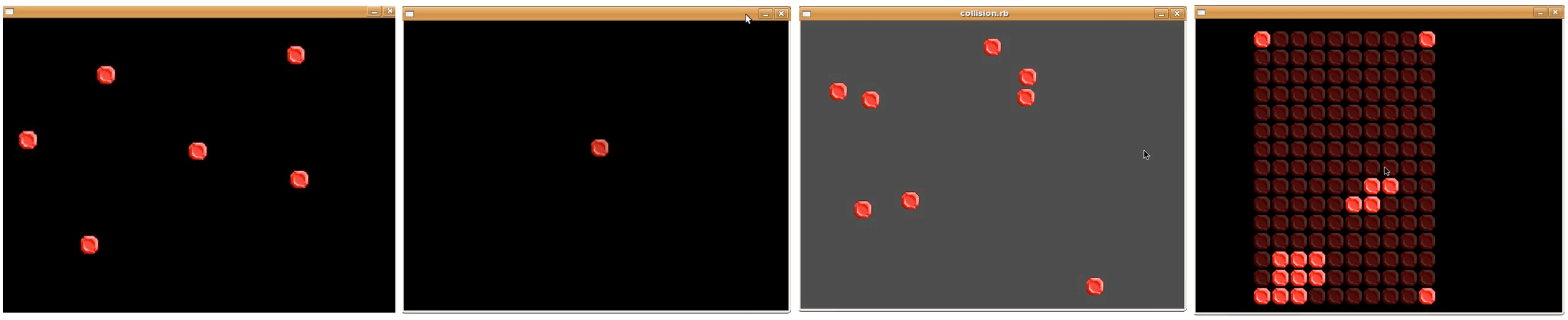
The one in the lower left demonstrates collision detection and handling. When 2 red shapes collide they are programmed to “bounce” off each other and change direction.

I thought SDL would be a good library to start with because it is the lowest level library I’m going to be covering and would give me an opportunity to go over game development concepts that apply to all the other frameworks.

We can look at some of these common functions right in the code of the samples apps that are packaged with Ruby\SDL. A number of these functions are encapsulated in the framework code for the other libraries.

Basically I’ll be pregressing through the libraries from lowest level to higher level.

Ruby/SDL Samples




```
screen = SDL::setVideoMode(640,480,16,SDL::SWSURFACE)
```

SDL::HWSURFACE

SDL::OPENGL

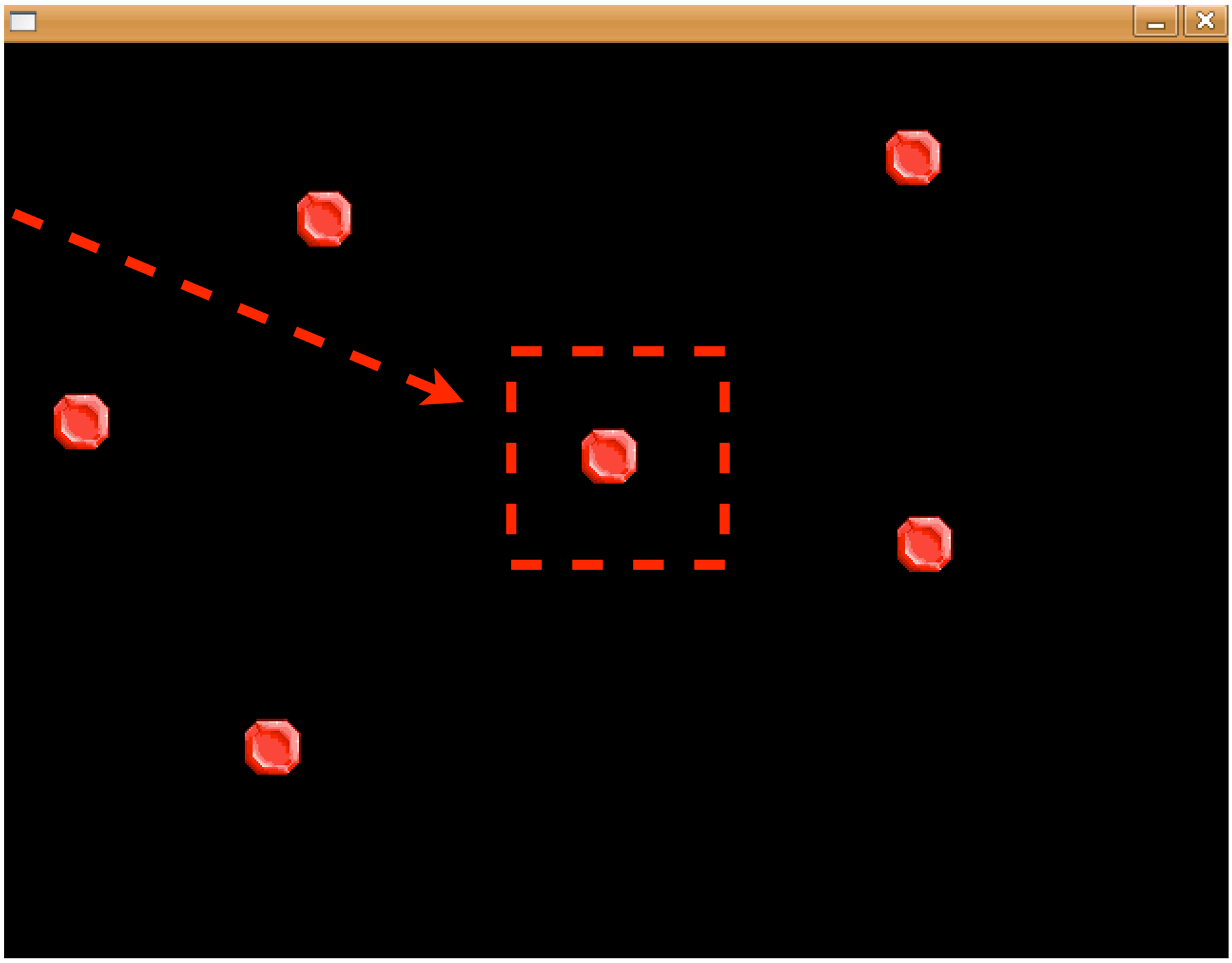
The setVideoMode method must be called for every Ruby/SDL app. It creates the display surface that the images will appear on.

You can pass it a flag to specify which rendering system to use: SW or Software surface, HW or hardware surface or OpenGL.

It's usually best do avoid HWSURFACE. Forcing HWAcceleration does not always improve performance. Not all ideo drivers are optimized in a way that complements SDL games. Also HWAcceleration is not an option on all platforms.

As for OpenGL, it's probably the fastest option -- but it's not very well integrated with Ruby \SDL. You would need to use the OpenGL API for drawing, which is not hard to understand, but can be somewhat cumbersome. We'll look at the OpenGL API a little later.

Ruby/SDL Sample

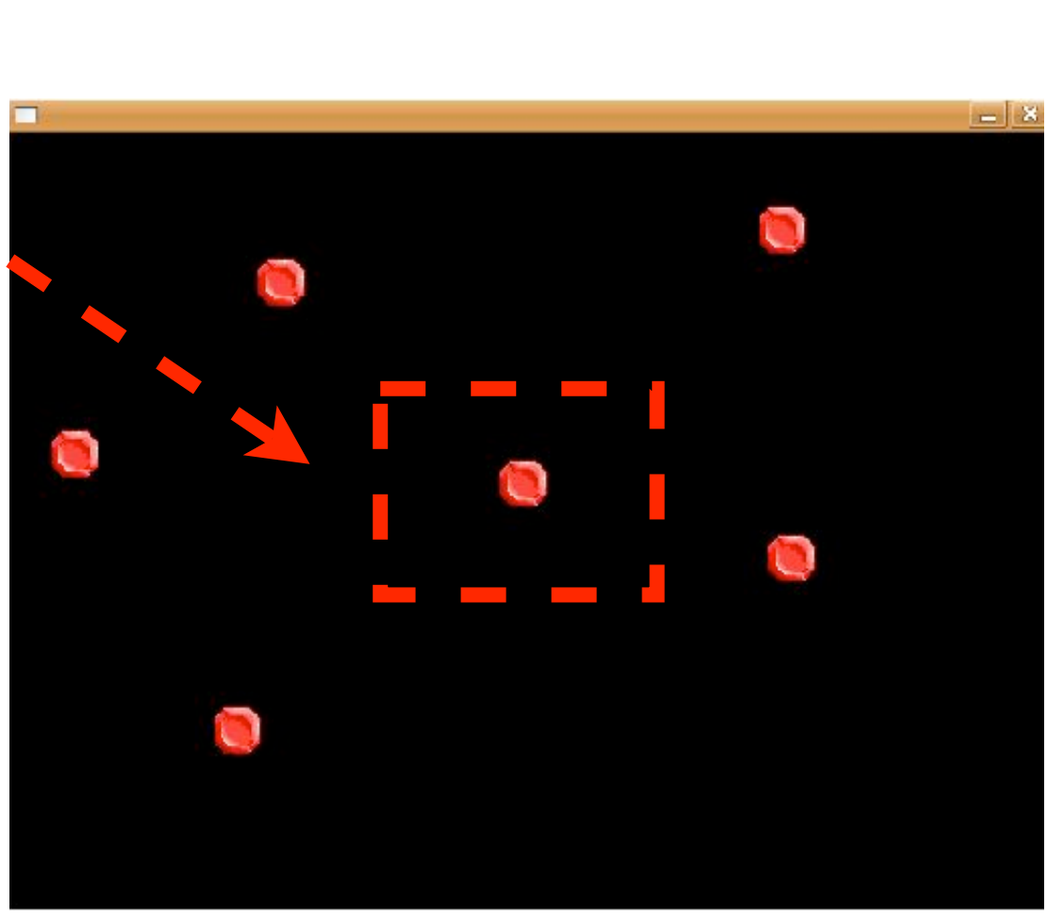


Now we'll take a closer look at the app with one controllable shape. As you'll see when we look at the source, you can't move it all around the screen. It gets sent back to the center of the screen at the beginning of each frame.

I think you'll agree that the code is very easy to follow.

Ruby/SDL Introducing Sprites

```
sprites = []
for i in 1..5
  sprites << Sprite.new
end
sprites << MovableSp.new
```

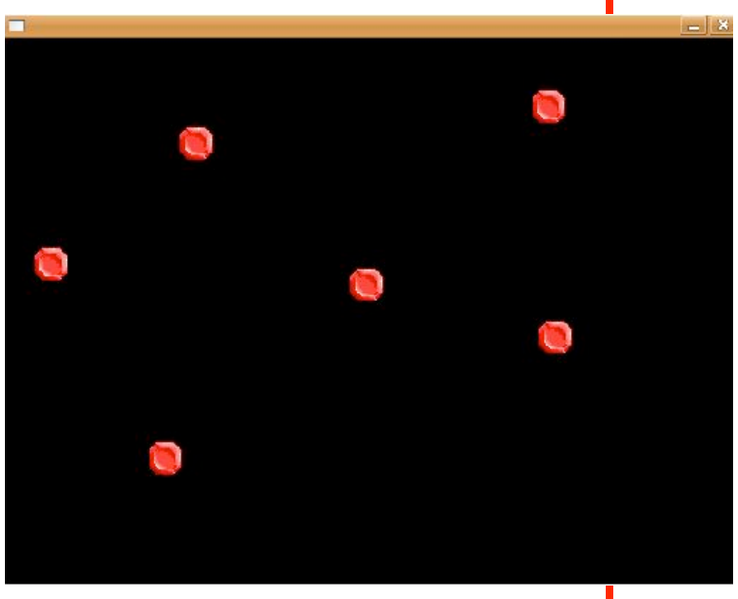


Here's the code that creates the all the red shapes. Sprite is the class that is responsible for ones that move randomly and MovableSp, short for MovableSprite, is the class responsible for the shape in the middle. A sprites array is created to hold all the sprites.

When I first started getting interested in video games, I thought a sprite was stock woodland elf in a fanstasy game. I'd see it used in contexts like this SDL game and wonder if the Sprites werent' supposed to represent elves dancing around. It turns out that "Sprite" is a standard computer graphics term for a 2D object whose position typically changes between frames.

Ruby/SDL Sample Event Loop

```
while true
  while event = SDL::Event2.poll
    case event
    when SDL::Event2::Quit
      exit
    when SDL::Event2::KeyDown
      exit if event.sym == SDL::Key::ESCAPE
    end
  end

  screen.fillRect(0,0,640,480,0)
  SDL::Key.scan

  sprites.each {|i|
    i.move
    i.draw(screen)
  }
  screen.updateRect(0,0,0,0)
end
```

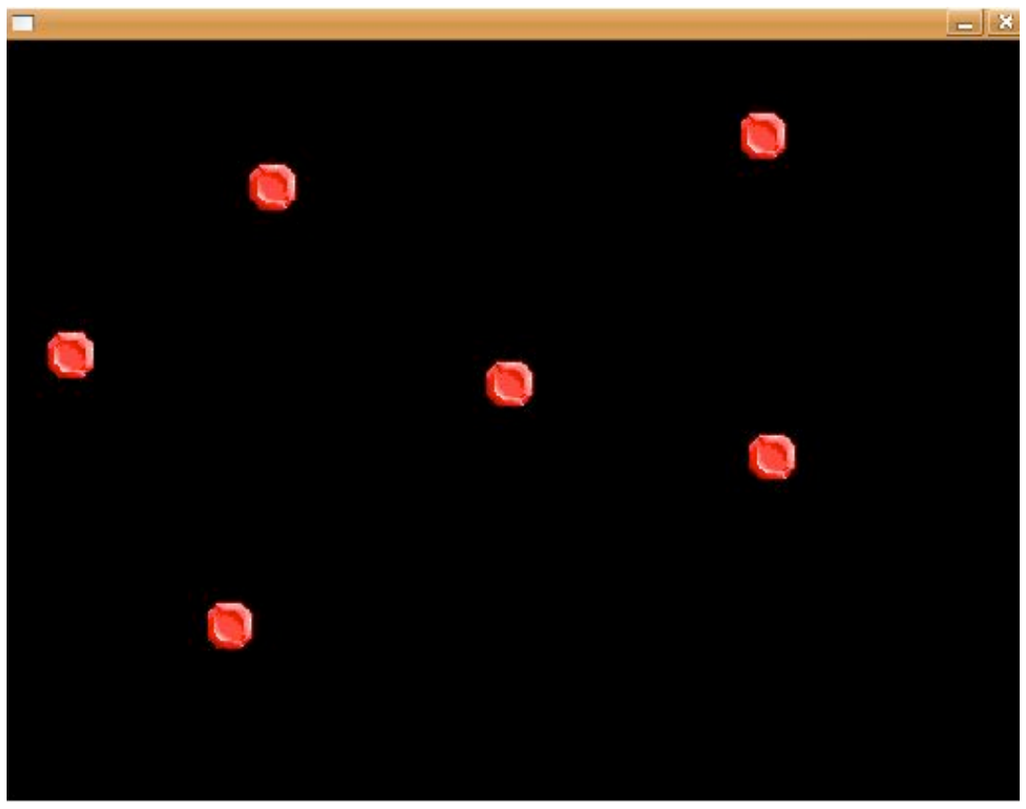


Here's the code that sets up a game loop and event queue processor. The Quit event is called when the user clicks on "x" button on the game window frame. So this event loop closes the app if the user hits ESCAPE or clicks the "x".

In each frame, move is called on all the sprites, then draw. Then the screen is updated by virtue of the updateRect call.
This is a pattern you will see over and over -- the sprites determine if and how they need to move or change their appearance, they are copied to the display surface in draw. Then the display gets updated to reflect the changes.

Ruby/SDL Sprite: Random Motion

```
class Sprite
  def initialize
    @x=rand(640)
    ...
    @dx=rand(11)-5
  end

  def move
    @x += @dx
    if @x >= 640 then
      @dx *= -1
      @x = 639
    end
    if @x < 0 then
      @dx *= -1
      @x = 0
    end
    ...
  end

  def draw(screen)
    SDL.blitSurface($image,0,0,32,32,screen,@x,@y)
  end
end
```



Here's the definition of the random motion sprites.

This is a pretty complete definition of the class -- but to save space, I took out the y code that corresponds with the x code.

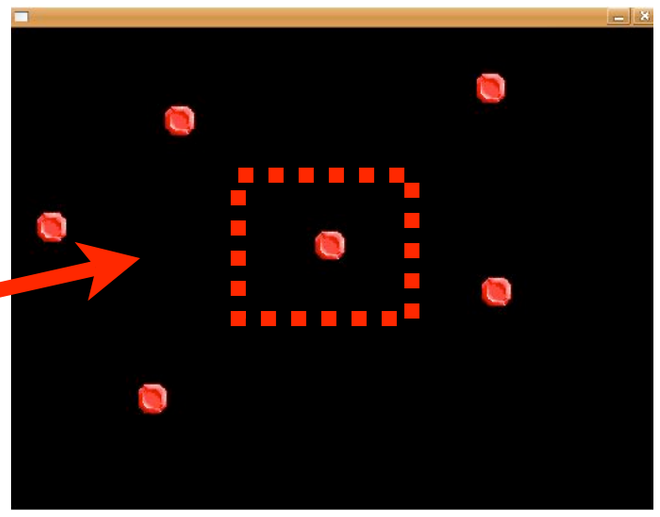
So in initialize, the Sprite's initial x and y coordinates are set randomly and dx and dy, which represent how much the sprite will move along the x axis and the y axis for each frame are set randomly.

The move method assigns values to the x and y coordinates, by either adding the dx and dy values or subtracting them if the sprite has reached an edge of the screen.

SDL.blitSurface, called in draw, copies the image of the red shape to the display surface.

Ruby/SDL Event-Driven Sprite

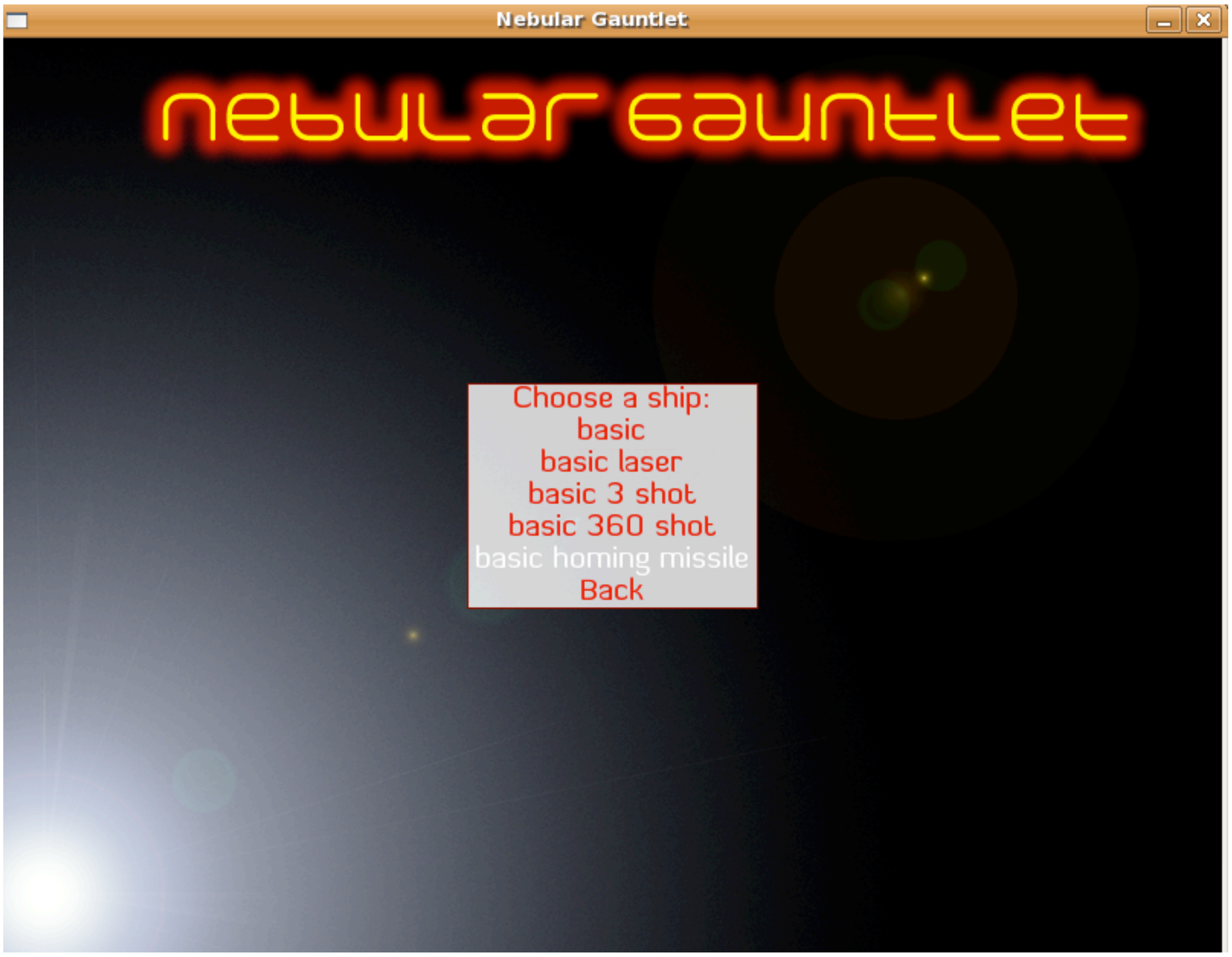
```
class MovableSp
  def initialize()
    @ud=@lr=0;
  end

  def move()
    @ud=@lr=0;
    @lr=-1 if SDL::Key.press?(SDL::Key::LEFT)
    @lr=1  if SDL::Key.press?(SDL::Key::RIGHT)
    @ud=1  if SDL::Key.press?(SDL::Key::DOWN)
    @ud=-1 if SDL::Key.press?(SDL::Key::UP)
  end

  def draw(screen)
    SDL.blitSurface($image,0,0,32,32,
    screen,300+@lr*50,200+@ud*50)
  end
end
```



Here's the Class definition for the sprite in the middle. When move is called it initializes both ud (up down factor) and lr (left right factor) to zero and then sets the values of ud and lr depending on which arrow key was pressed, if any.

In draw, the last two parameters represent the new x and y coordinates for the sprite and are highlighted in red. When both factors are zero, the sprite is drawn at the center of the screen. Otherwise the Sprite moves 50 pixels or negative 50 pixels from center in the specified direction.

Ruby/SDL Steven Davidovitz's Nebular Gauntlet



Project Home: http://www.nebulargauntlet.org/

Now we'll look at actual game that was written using Ruby/SDL.

Steven Davidovitz's Nebular Gauntlet is a great resource for learning about SDL techniques, like scrolling the background and using a particle engine to simulate rocket fire. It's still a work in progress, but there are already a lot of features in place. I really like this little radar area that shows all the entities on the screen.

You can save games. It comes with a map editor that lets you create your own maps in point and click fashion. With the map editor you can specify where the shields and bots should be placed. Bots are programmed to chase your ship.

Ruby/SDL Steven Davidovitz's Nebular Gauntlet

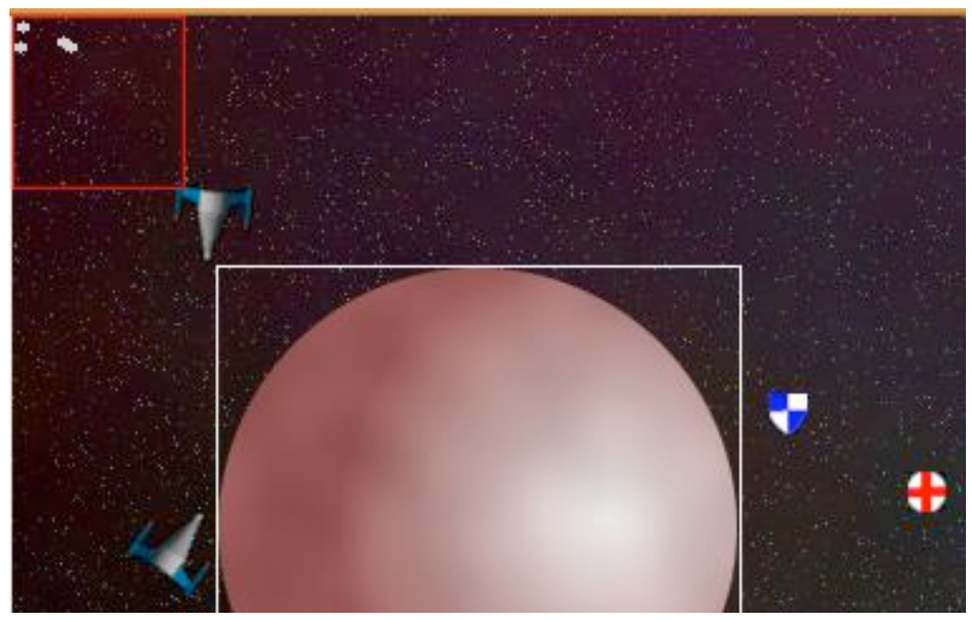
```
while !@@quit
  handle_input() # Handle keyboard input
  do_think() # Do calculations/thinking
  do_render() # Render everything
  Signal.trap("INT", @on_quit)
  Signal.trap("EXIT", @on_quit)
end
```

Project Home: http://www.nebulargauntlet.org/

Even though it has so many features, it's still structured exactly like the rudimentary samples game we looked at. Here's the main event loop. It evaluates input -- determines if the ship of other entities need to change position and repositions them -- and then updates the display on the screen.

Ruby/SDL Steven Davidovitz's Nebular Gauntlet

```
class Radar < Box
  def initialize(screen, size_modifier, *objects)
    @size = size_modifier
    height = $map.h / @size
    width = $map.w / @size
    super(screen, height, width)
    @objects = objects
  end

  def draw
    @screen.drawRect(@x, @y, @width, @height, @bcol)
    @objects.each do |obj|
    @screen.drawFilledCircle(0 + (obj.x / @size,
         0 + (obj.y / @size), 2, [200, 200, 200])
         if obj.state == :alive
    end
  end
end
```



Project Home: http://www.nebulargauntlet.org/

Here's the code that's responsible for the radar area. It loops through all the ships, bots and shields and scales a proportional model of the action by dividing the x and y coordinate of each object by a size modifier -- which is set as 15 in the application initialization code. Then is draws a white circle to represent each one.

Danny Van Bruggen, Creator

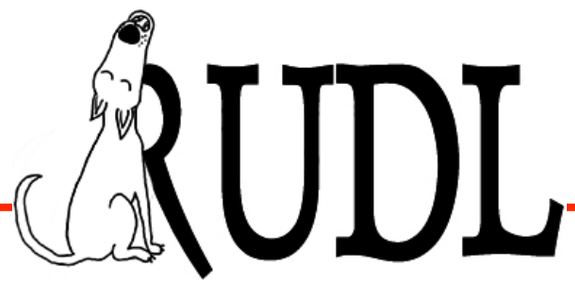
http://sourceforge.net/projects/rudl/

There has not been active development on the project for a few years, but it's a good source of ideas and its packaged with some interesting resources.

RUDL was created to provide a way to use SDL with more conventional Ruby syntax and style and also to minimize the amount of boilerplate code the user has to write, like initialization code.

Principle of Least Surprise?

```
SDL_Surface *SDL_SetVideoMode(int width, int height,
                              int bpp,Uint32 flags);
```



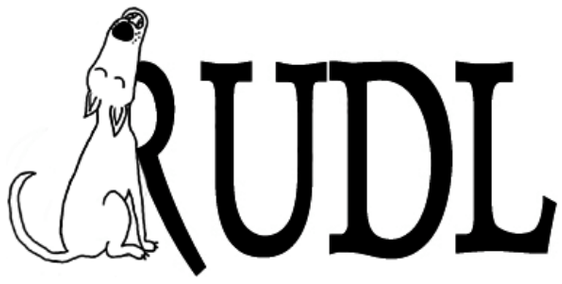

```
static VALUE displaySurface_new(int argc, VALUE* argv,
                                VALUE self)
```

The RUDL DisplaySurface exemplifies RUDL's approach to meeting those goals.

It's the RUDL wrapper for the C SDL library's setVideo Mode, which creates the display surface that the images will appear on.

RUDL creator Danny Van Bruggen felt that it violated the principle of least surprise for a method that creates a new display surface and returns it to be called setVideoMode. He thought you would expect a DisplaySurface to be returned if you create new DisplaySurface.

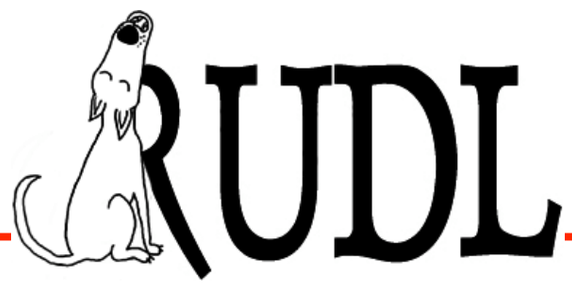
He also didn't think it was necessary to pass 4 parameters to it, when the last 3rd parameter represents bits per pixel and is likely to be set at 16, and the last parameter represents the redning engine, whcih is likely to be SWSURFACE.



Principle of Least Surprise?

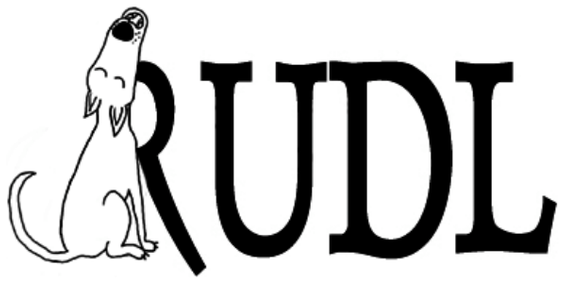
Ruby/SDL

```
screen = SDL::setVideoMode(640,480,16,SDL::SWSURFACE)
```



```
display = DisplaySurface.new([640,480])
```

So here's a sample RUDL call to the setVideoMode wrapper as compared to a sample Ruby \SDL call to it's setVideoMode, which is a very thin, literal wrapper around the SDL C library's setVideoMode.



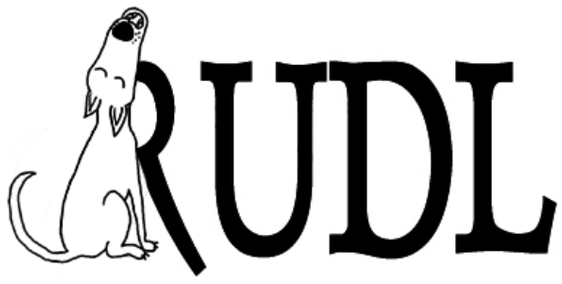
Principle of Least Surprise?

Ruby/SDL

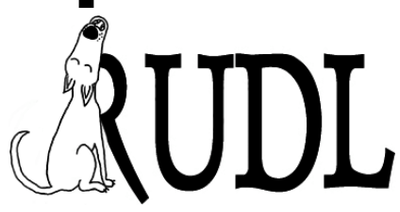
```
static VALUE sdl_setVideoMode(VALUE mod,VALUE w,VALUE h,
    VALUE bpp,VALUE flags){
  SDL_Surface *screen;
  screen = SDL_SetVideoMode(NUM2INT(w),NUM2INT(h, NUM2IN
           (bpp), NUM2UINT(flags));
  if(screen == NULL){
    rb_raise(eSDLError,
      "Couldn't set %dx%d %d bpp video mode:%s",
      NUM2INT(w),NUM2INT(h),NUM2INT(bpp),SDL_GetError());
  }
  return Data_Wrap_Struct(cScreen,0,0,screen);
}
```

```
rb_define_module_function(mSDL, "setVideoMode",
sdl_setVideoMode,4);
```

Here's the complete source for the Ruby\SDL Wrapper. It's a very thin, literal wrapper that does the minimum required to enable setVideoMode to be called from a Ruby app.



Principle of Least Surprise?

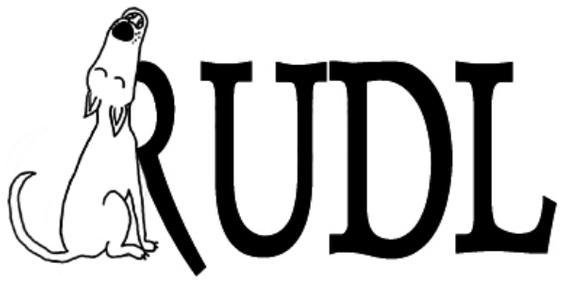


```
static VALUE displaySurface_new(int argc, VALUE*
argv, VALUE self)
{
   ...
   surf = SDL_SetVideoMode(w, h, depth, flags);
   currentDisplaySurface = Data_Wrap_Struct
     (classDisplaySurface, 0, 0, surf);
   return currentDisplaySurface;
})
```

```
rb_define_singleton_method(classDisplaySurface,
                           "new",
                           displaySurface_new,
                           -1);
```

Here's the corresponding RUDL wrapper. The elipsis is a place holder for the many things that happen in that wapper, like evaluating which of the optional parameters were passed in and the pyrotechnics necessary to make the C extension wraper behave like a constructor for the DisplaySurface class.

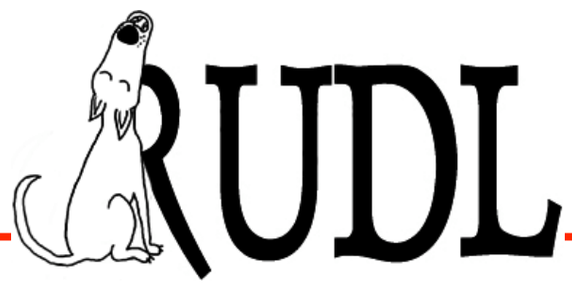
Sometimes instead of putting a lot of code in the C extension files, developers create thin wrappers for the C library functions and then write a pure Ruby library to make the API more elegant and add functionality.



Principle of Least Surprise?

Ruby/SDL

```
screen = SDL::setVideoMode(640,480,16,SDL::SWSURFACE)
```



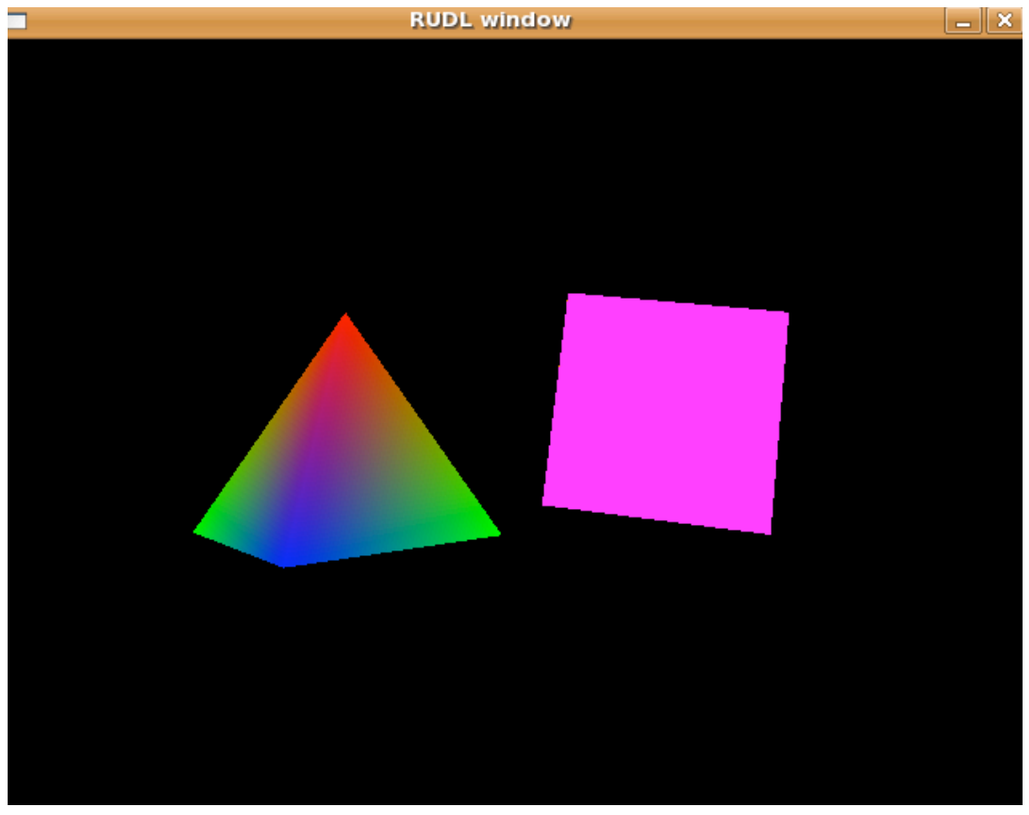
```
display = DisplaySurface.new([640,480])
```

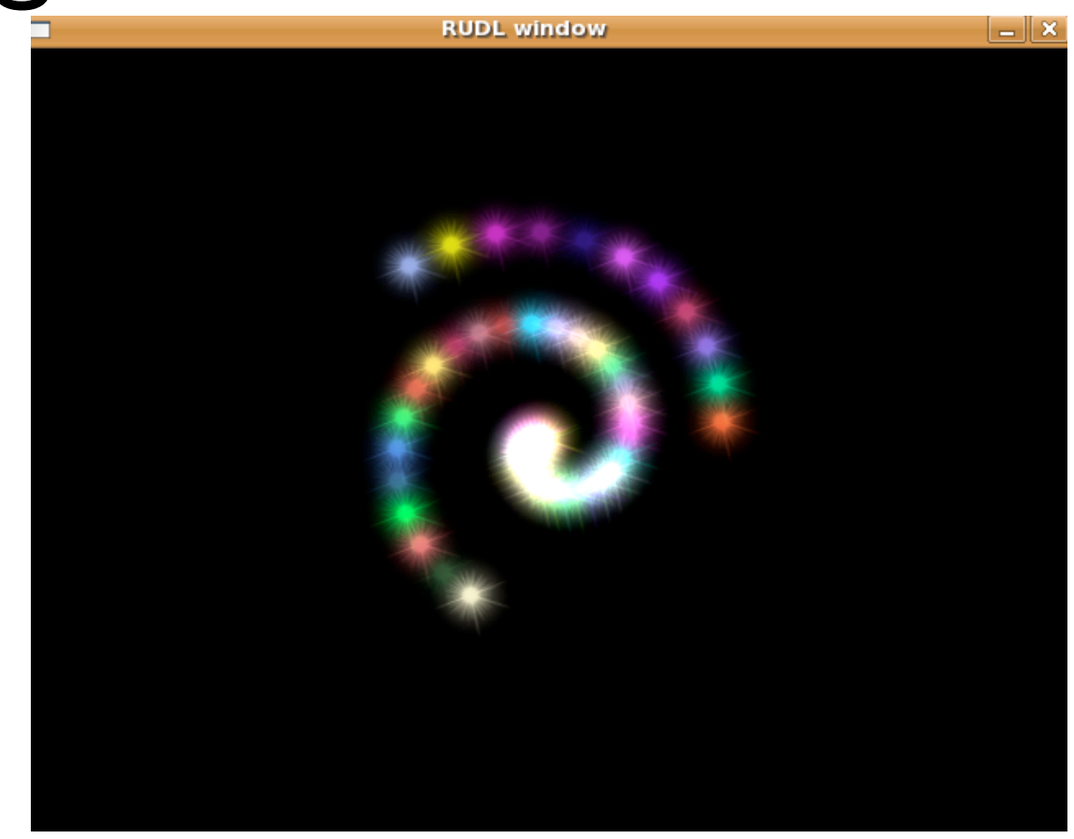
Here's that side by side comparision again. Arguably it's less surprising for a call to DisplaySurface.new to create and return a new display surface than a call to a method called setVideoMode.

But there's also a case to be made for there being a sense in which the Ruby\SDL wrapper is less surprising **as an SDL wrapper.** One advantage to a thinner more literal wrapper is that they make it easier for users of the extension library to use the original library documention.

RUDL OpenGL

http://nehe.gamedev.net/





RUDL Ports of Neon Helium Tutorials, Packaged with RUDL, Ported by Martin Stannard

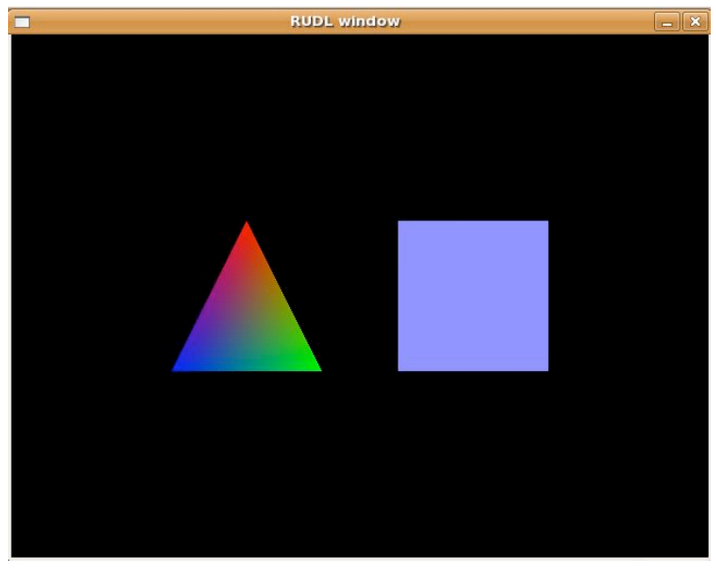
One of the very useful resources packaged with RUDL is a set of Ruby translations of the Neon Helium OpenGL tutorials.

These translations use OpenGL as the rendering system, but use RUDL for everything else, like event handling.

While we're talking about RUDL, then I want to show what the OpenGL API looks like.

RUDL OpenGL

```
def init(w,h)
  GL.ClearColor(0.0, 0.0, 0.0, 0.0)
  GL.ClearDepth(1.0)
  GL.DepthFunc(GL::LESS)
  GL.Enable(GL::DEPTH_TEST)
  GL.ShadeModel(GL::SMOOTH)
  GL.MatrixMode(GL::PROJECTION)
  GL.LoadIdentity()
  GLU.Perspective(45.0, w.to_f/h.to_f,
     0.1, 100.0)
  GL.MatrixMode(GL::MODELVIEW)
  GL.LoadIdentity()
  GLU.LookAt(0.0, 0.0, 5.0,
    0.0, 0.0, 0.0, 0.0, 1.0,
    0.0)
end
```

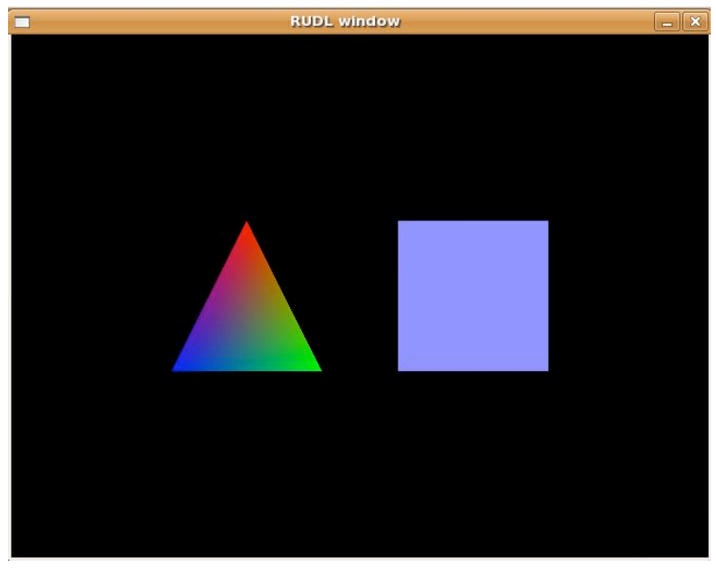


Source by Martin Stannard: from RUDL samples

This code is from a sample that creates a couple of 2 dimensional primatives. First using OpenGL requires a fair amount of initializaion.

RUDL OpenGL

```
display = Proc.new {
GL.Clear(GL::COLOR_BUFFER_BIT|
         GL::DEPTH_BUFFER_BIT)
GL.Color(1.0, 1.0, 1.0)
GL.LoadIdentity()
GL.Translate(-1.5, 0.0, -6.0)
# draw a triangle
GL.Begin(GL::POLYGON)
GL.Color3f(1.0, 0.0, 0.0)
GL.Vertex3f(0.0, 1.0, 0.0)
GL.Color3f(0.0, 1.0, 0.0)
GL.Vertex3f(1.0,-1.0, 0.0)
GL.Color3f(0.0, 0.0, 1.0)
GL.Vertex3f(-1.0,-1.0, 0.0)
GL.End()

GL.Translate(3.0,0.0,0.0)
}
```



Source by Martin Stannard: from RUDL samples

Here's the code that renders the triangle.

Since OpenGL is a 3D rendering system, points are represented by vertexes which have an x coordinate, a y coordinate and a z coordinate. The z axis goes from front to back. 0 is on the screen surface. If you're sitting in front of a terminal, front pops out of the screen towards you and back is going back into virtual space behind the screen. To make an object appear to go into the screen you decrease it's z value.

A 2D scene, like this, can be created with OpenGL by always setting the z coordinate to 0. Setting the triangle's 3 vertex's requires 3 glVertex calls, and setting a color for each vertex requ
requires 3 GL color calls.

Drawing the triangle requres GLBegin and GLEnd to group those vertexes.

Creating a 3D pyramind would require 3 times as many GL::vertex calls to define the 3 surfaces.
To display an image instead of coloring the shape you would use glBindTexture and then you would use glTexCoord calls in lieu of the GL Color Calls.

Creator and Lead Developer: John Croisant

http://rubygame.sourceforge.net/info.html

Rubygame is a full-featured, high-level game development library that also exposes its lower-level wrappers around SDL's C API.

You can jumpstart your project using Rubygame's convenience methods and helper classes -- but if necessary, you can access SDL functions through Rubygame on an "a la carte" basis to tweak your code.

Rubygame was initially modeled after Pygame, a popular Python-based framework -- and its name reflects that. According to John Croisant, the creator of Rubygame, he chose Pygame because it was the best game development framework he knew about at the time.

Over time, even the features that come closest to being direct ports from Pygame have become more distinctly Ruby-tinged in the way they are implemented.

Rubygame 3.0, the next major release of the framework, will be a radical departure from Pygame. John is working to ensure that older games will run with the new system, though some adjustments may be required.

Before I go into some of what's in store with Rubygame 3.0, I want to give you a feel for what it means for a game engine to be Pygame-like by looking at some of the sample apps packaged with Rubygame 2.0.1, the current version of Rubygame.

Rubygame
Game programming should be fun.

and

pygame

Sample Packaged with Rubygame

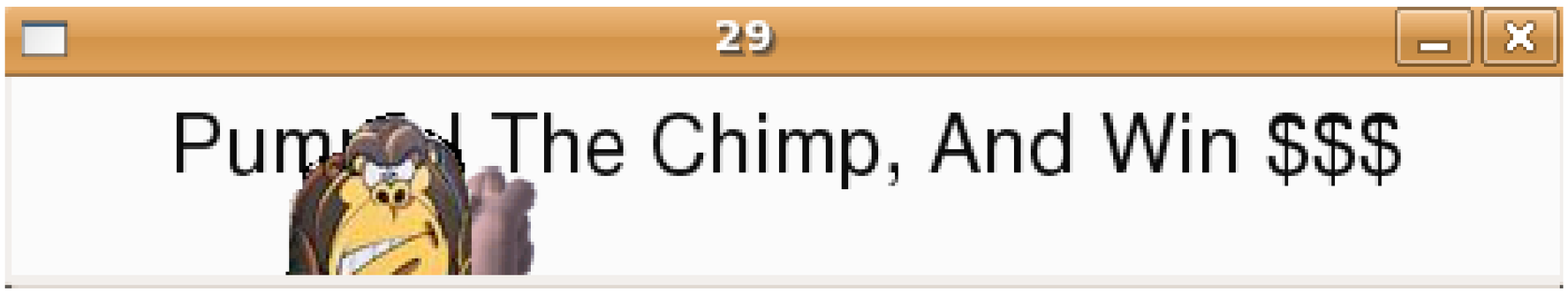
This Punch the Chimp game that resembles a banner ad is a direct port from a Pygame tutorial. The Rubygame and Pygame APIs are similar enough that it's very easy to follow along in the Rubygame code while going through the step-by-step Pygame tutorial.

Before we see this game in action, I'm just going to substitute a picture of a gorilla for the original chimp. It sounds silly, but I couldn't bring myself to smack a realistic-looking chimp.

Rubygame
Game programming should be fun.

and

pygame



Sample Packaged with Rubygame

You move the fist with the cursor, and throw your punch by clicking the mouse button. If you make contact, the chimp spins.

Rubygame
Game programming should be fun.

and

pygame

Rect

Rect, as in “Rectangle”, is one of the main classes in Pygame, and likewise one of the main classes in Rubygame 2.x.

Rects are typically paired with Sprites, and are usually based on the dimensions of the image that represents the Sprite.

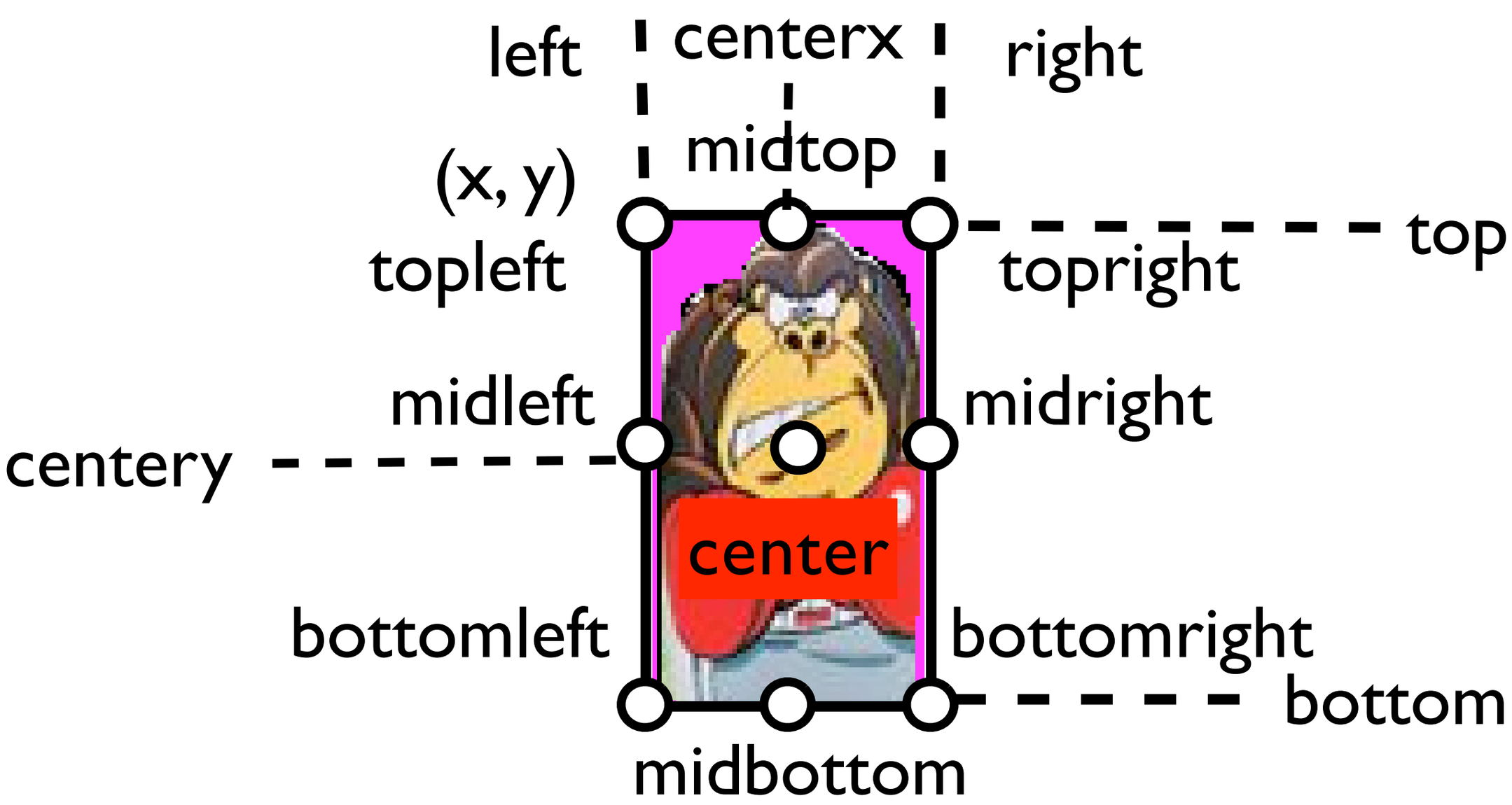
Here is the bitmap that represents the Chimp in the game. Game frameworks typically allow the user to specify a color that should not be rendered when a particular image is displayed -- in this case, fushia. Without the background the chimp cuts a fine figure in the app.

Rubygame
Game programming should be fun.

and

pygame

Rect




```
@rect.topleft = 10,10
```

Rects can be used to move Sprites.

You can position a Sprite by setting any of the attributes represented here by circles or dashed lines on the Sprite's Rect. The snippet in the red box is used to position the Chimp when the Punch the Chimp game starts.

The fist's Rect's midtop value is set to match the mouse's coordinates in order to make the fist follow the mouse.

Rubygame
Game programming should be fun.

and

pygame

Rect


```
# Attempt to punch a target.
# Returns true if it hit or false if not.
def punch(target)
  @punching = true
  return @rect.inflate(-5,-5).collide_rect?(target.rect)
end
```

The Rect class also provides collision detection services, and a lot of utility methods like clamp, which puts a Rect right inside another Rect, and inflate, which can make the Rect grow or shrink depending on whether you pass in positive or negative numbers.

This is the code that determines whether the fist has made contact. The test is made with a smaller version of the fist's Rect, made by deflating it -- or inflating it with negative parameters, to ensure that a punch only registers if the fist hits its target squarely.

Rubygame
Game programming should be fun.

and

pygame

Sprites::Sprite, Sprites::Groups, Sprites::UpdateGroups

Sample Packaged with Rubygame

NOTE: THIS IS A MOVIE, NOT A STATIC SCREEN SHOT.

Another utility class that is central to Pygame programming is the Sprite.group -- and the Group class in the Sprite module is likewise important for Rubygame.

Sprites::Groups handle bulk actions for their constituent Sprites, including drawing, updating and collision detection. In this Rubygame sample app, which shows how to achieve a number of different effects with Rubygame, the Pandas all belong to the same Sprites::Group. Often in real games, the Sprites in a scene do not belong to the same Sprites::Group. Games can organized around SpriteGroups. There can be different SpriteGroups for different teams for example. SpriteGroups are easily extensible.

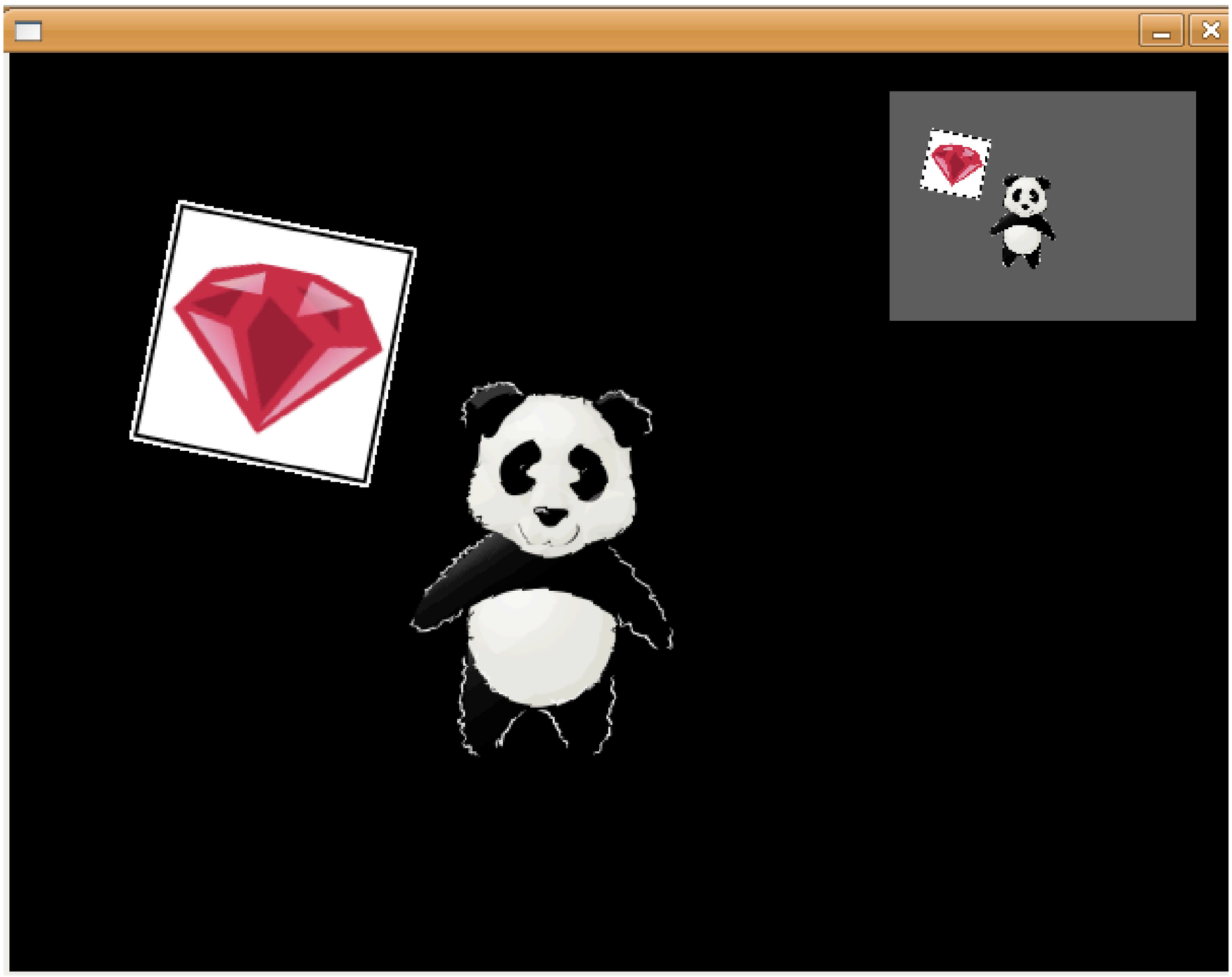
When we looked at the RubySDL sample code, we saw that simple group updates can be achieved by putting all the Sprites in an Array and looping through the Array to redraw each Sprite every frame. So what makes Sprites::Groups special? One example of a useful feature, which is available if you mix in Rubygame's UpdateGroup module or use Pygame's RenderUpdate module, is that they can keep track of the Rects that were repositioned since the last update and only redisplay those.

Edge

Now we'll look at some of what's going on in the development branch. One of the major changes is the new scene management system, replete tight with OpenGL integration, a new event management system, and a positioning system that is very different from the Pygame-like rect-based one.

Edge Rubygame Scene Management

Rubygame
Game programming should be fun.



Sample Packaged with the Development 3.0.0 branch

NOTE: THIS IS A MOVIE NOT A STATIC SCREEN SHOT

Here's a demo app from the Rubygame 3.0.0 development branch.

The big panda follows the mouse. When you click on the screen both the panda and the ruby jump to the cursor. When the panda and the ruby collide, they turn red.

Edge Rubygame and OpenGL

```
ruby = GLImageSprite.new {
  @surface = Surface.load_image('ruby.png')
  setup_texture()
  @pos = Vector2[100,300]
  @depth = -0.1
  @angle = -0.2
}
```

Here's the code that sets up the Ruby Sprite in the new system. Behind the scenes it's using the same sort of bulky sequence of OpenGL API calls we looked at in the RUDL samples, but here they are wrapped with a single setup_texture call.

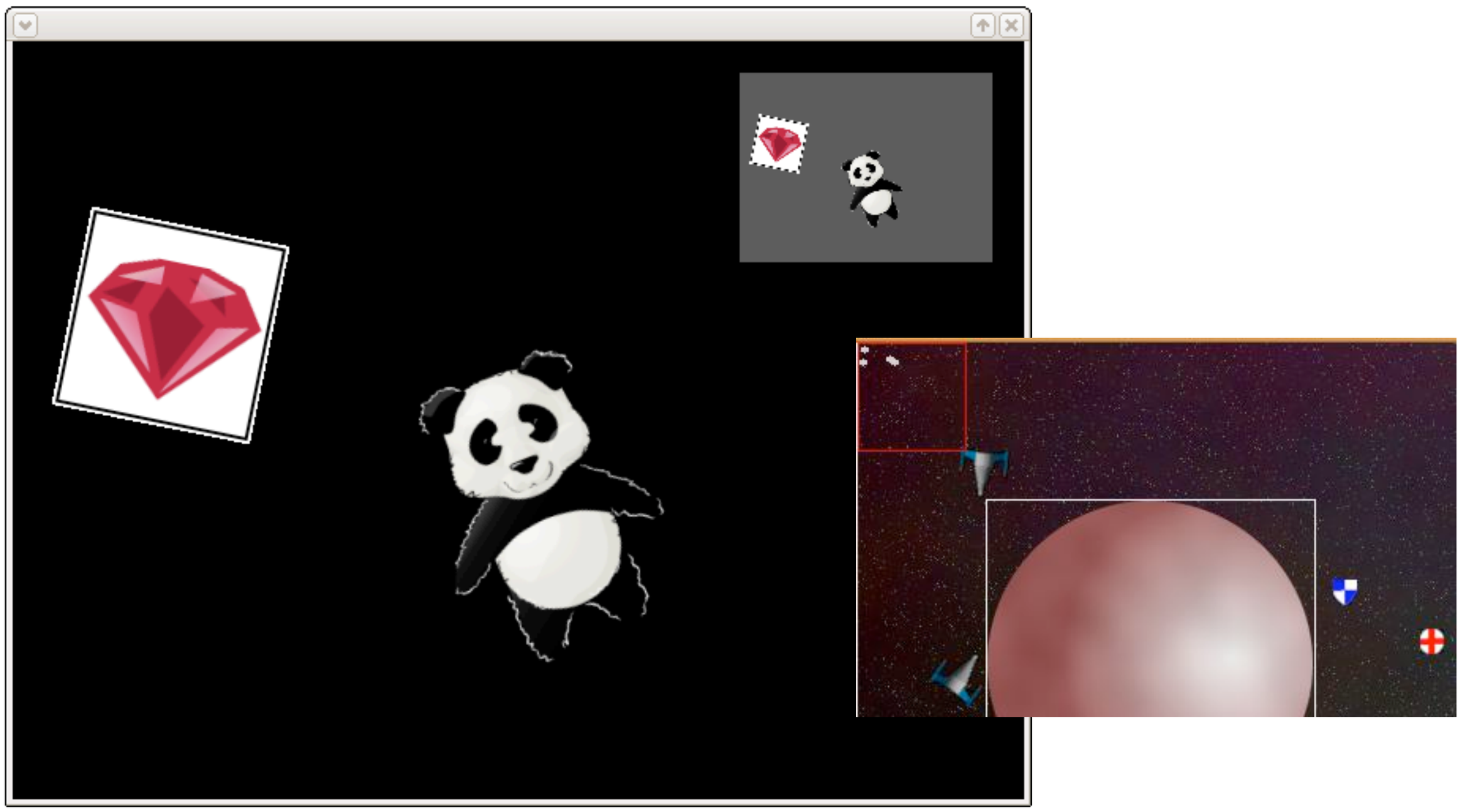
The depth variable enables the user to specify whether a particular Sprite should be rendered closer to the front or the back of a scene in relation to the other Sprites. The enhanced Rubygame 3.0 SpriteGroups will be responsible for drawing the Sprites in the proper order.

The position is set using two coordinates, but behind the scenes, the framework is using OpenGL's 3D positioning system with the "z" coordinate set to 0.

There's tight OpenGL integration in the Rubygame 3.0 branch, but John is committed to making the new system work without requiring a 3D graphics card. There will be an alternative implementation of the new scene management framework that will not require Open GL.

Edge

Rubygame
Game programming should be fun.



The Rubygame picture-in-a-picture may remind you of the little radar screen we looked at in Nebular Gauntlet but the implementations are entirely different.

In Nebular Gauntlet the scaled-down version is achieved by drawing a small shape to represent each space ship or shield.

In this Rubygame 3.0 demo app, the window in the upper right is showing another view of the scene by virtue of a virtual camera with a perspective that differs from the scene manager's default perspective.

You can think of a software camera as being similar to a cell-phone camera in that both involve focusing on a region of the world -- in the software realm this where the objects live -- and projecting it onto a 2-dimensional screen. If the world region is defined to match the default camera's world region, but it's screen region is smaller, as it is in this case, the figures appear diminished.

There's no application code that draws a second panda or ruby, like the white circles that have to be drawn onthe radar screen in Nebular Gauntlet. The second camera just needed to be added using add_camera.

A release date for Rubygame 3.0 has not been scheduled yet. There are still a lot of ideas that John would like to incorporate into it.

He has ideas that push the envelope of game development that you can read about in the comments in the Rubygame code and also in his blog. Here's a recent entry about why the RGB color model is inadequate for rendering a scene in the middle of a tunnel lit with yellow. He suggests that many developers would just tint everything in the scene yellow, and considers what might be involved in making the scene more true-to-life.

In real life, the limited spectrum emitted by the lights in the tunnel would make a red car appear to be a dark yellow-orange or a blue car appear nearly black.

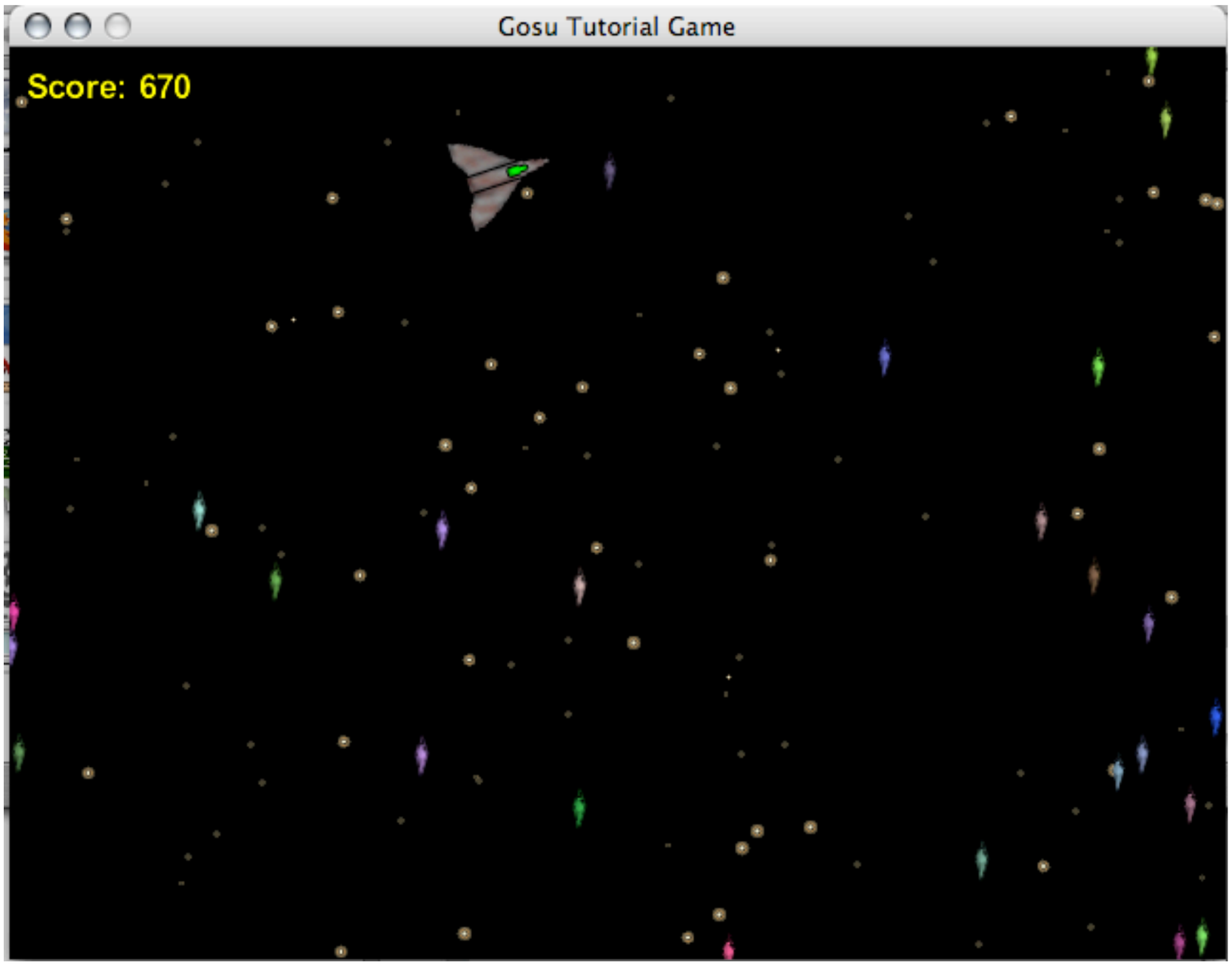
Co-Creators: Julian Raschke & Jan Lücker
Lead Developer: Julian Raschke

http://code.google.com/p/gosu/

Gosu is polished and compact. Its development team aims to include everything you need to create a game, but nothing more. Every feature it supports was needed for and tested in an existing application. Nothing was added because someone might find it useful in theory.

In keeping with the design rationale behind the code, Gosu is packaged with a well-thought-out tutorial that provides everything you need to get started with Gosu, but nothing more. Therefore, it takes less than an hour to complete, but at the end of that hour you're ready to start developing games.

Tutorial



Here's what the finished tutorial looks like. When you steer the ship into a star, the star vanishes and you get points. It was not designed to be challenging. It was expressly designed as teaching tool.

NOTE: THIS IS A MOVIE, NOT A STILL SCREEN SHOT

Main Window

```
class MainWindow < Gosu::Window
  def initialize(width, height, fullscreen)
    super
  end

  def update
    # Change display attributes, if necessary.
  end

  def draw
    # Display calls go here.
  end

  def button_down(id)
    # Buttons are keys, mouse buttons, and
    # game pad controls.
  end
end

w = MyWindow.new(640, 480, false)
w.show
```

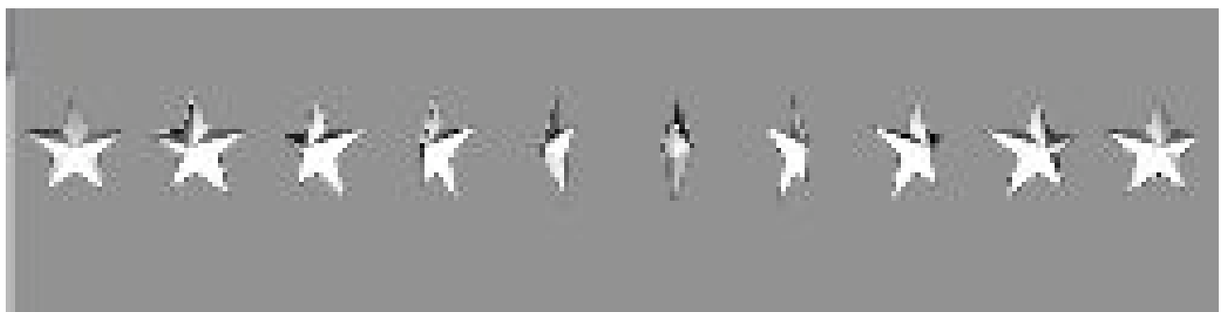
Even experienced Gosu developers start out by subclassing Gosu's Window class and taking advantage of its built-in services. Here’s a skeletal main window.

You don't need to start a main game loop or set up an event queue in your application-level code because Gosu handles that behind the scenes. Gosu will call the main window’s update() method and then its draw() method, every frame. When the user presses a key on the keyboard or a mouse button, the button_down() callback is invoked.

By the end of the tutorial, the main window’s update() implementation calls methods on the spaceship and the stars to determine if and how they need to be repositioned or modified before the next time the screen is redisplayed, and repositions them if necessary.

Techniques

```
class Star
  attr_reader :x, :y

  def initialize(animation)
    @animation = animation
    @color = Gosu::Color.new(0xff000000)
    @color.red = rand(255 - 40) + 40
    @color.green = rand(255 - 40) + 40
    @color.blue = rand(255 - 40) + 40
    @x = rand * 640
    @y = rand * 480
  end

  def draw
    img =
      @animation[Gosu::milliseconds/100 % @animation.size]
    img.draw(@x - img.width/2.0, @y - img.height/2.0,
        ZOrder::Stars, 1, 1, @color, :additive)
  end
end
```



Here's a quick look at how the Stars are animated. Each star gets a random position and color when it is created. About ten times a second, each Star's image shifts to the next image in an image array created from the .png file shown here.

NOTE: THE STAR INSET WILL BE A LITTLE MOVIE, SHOWING THE STAR ANIMATION.

Nightly Travels of a Witch

Team Tortilla: Florian Gross, Julian Raschke, Alexander Post

Because Gosu is so streamlined, many developers have found that its well-suited for timed game development competitions. This game was created for a competition by a team of developers, including Julian Raschke and Florian Gross, who posted a link to it on the ruby-talk mailing list. It was created in 72 hours, with at least some team members taking time out for sleeping during the event. The version shown here was polished a little after the competitions ended.

The object of this game is to lure the sleepwalking witch back to bed, primarily using chocolate as bait. The fairy is actually the mouse cursor, and if you click on the chocolate bar, a tiny chocolate bar will appear in her hands. If you feed her a hot chili and scare her with a plush Orc toy in her sleep, she becomes red-tinted and infused with firepower that enables her to zap anything the gets in her way. Feed her an ice cream cone when she needs to throw ice bolts to neutralize fiery cauldrons and create ice bridges out of pools of water she needs to cross. On some levels you need to scare her with a spider, so she will hit a light switch when she tries to swat it away, shutting off a light that could potentially wake her up.

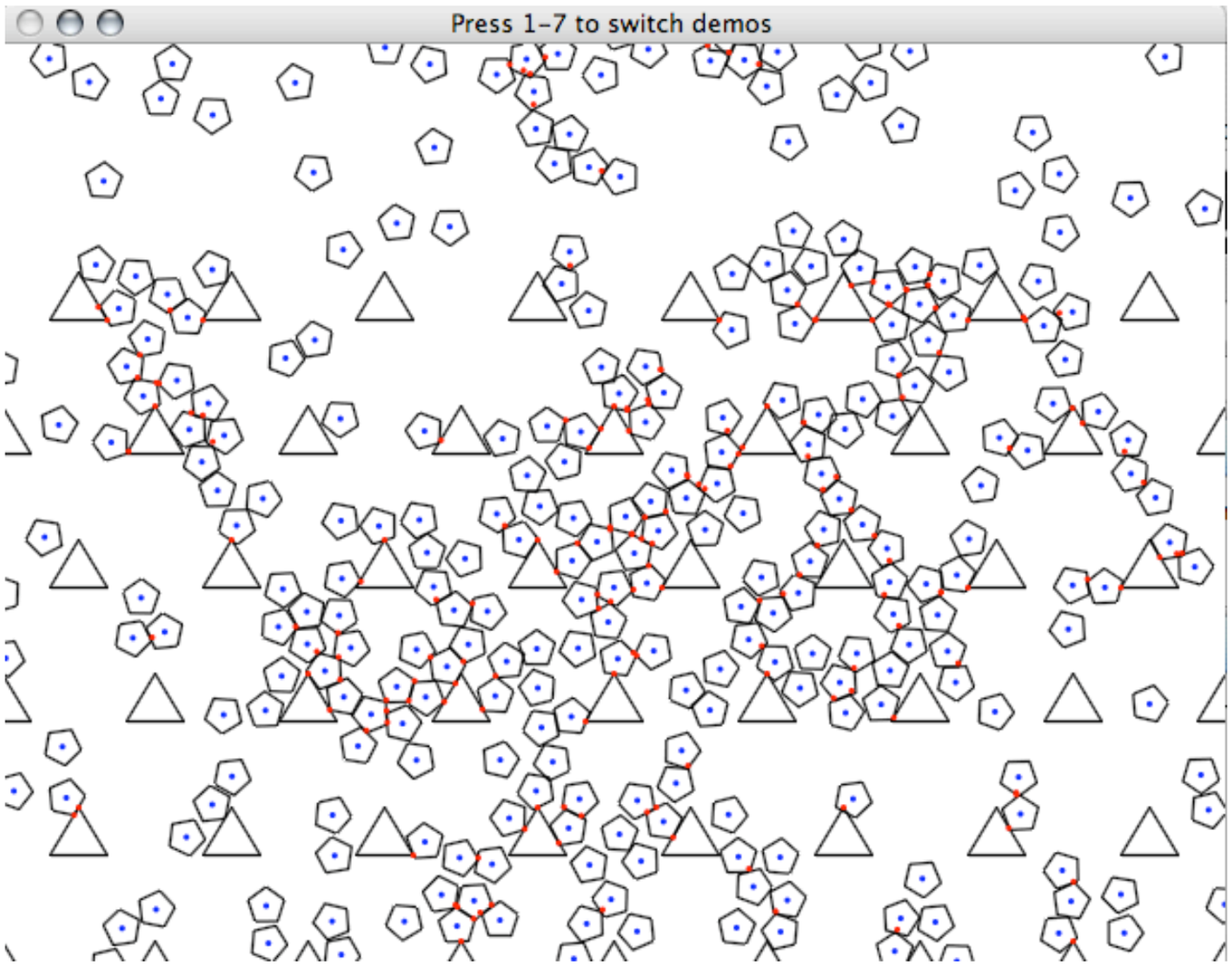
NOTE: THIS IS A MOVIE, NOT A STILL SCREEN SHOT

and

CHIPMUNK
GAME DYNAMICS

wiki.slembcke.net/main/published/Chipmunk

Creator and Developer: Scott Lembcke



Sample Packaged with Chipmunk

Part of the Gosu philosophy is that anything that is not essential to for most games should be added by integrating with additional libraries.

As part of a recent Gosu release, an tutorial that shows how Gosu can be integrated with the Chipmunk 2D physics library was added. Chipmunk is written in C, but is packaged with its own Ruby bindings.

Chipmunk enables you to you imbue your Sprites with virtual mass, so that when you apply virtual force to them, and when they collide -- they behave according to the laws of natural physics.

NOTE: THIS IS A MOVIE, NOT A STILL SCREEN SHOT

and


```
if rand(100) < 4 and @stars.size < 25 then

  body = CP::Body.new(0.0001, 0.0001)
  shape = CP::Shape::Circle.new(body, 25/2,
    CP::Vec2.new(0.0, 0.0))
  shape.collision_type = :star
  @space.add_body(body)
  @space.add_shape(shape)

  stars.push(Star.new(@star_anim, shape))
end
```

Gosu\Chipmunk Tutorial by Dirk Johnson

The Chipmunk tutorial uses the original Gosu tutorial we looked at a few minutes ago as a starting point.

This code block shows how each star is linked to a body, which is in turn is linked to a shape.

A Rigid Body in Chipmunk has physical properties like mass and velocity, while a shape represents
an area on the screen. The shape associated with a sprite is inspected at collision time to see if it overlaps another entity's shape.

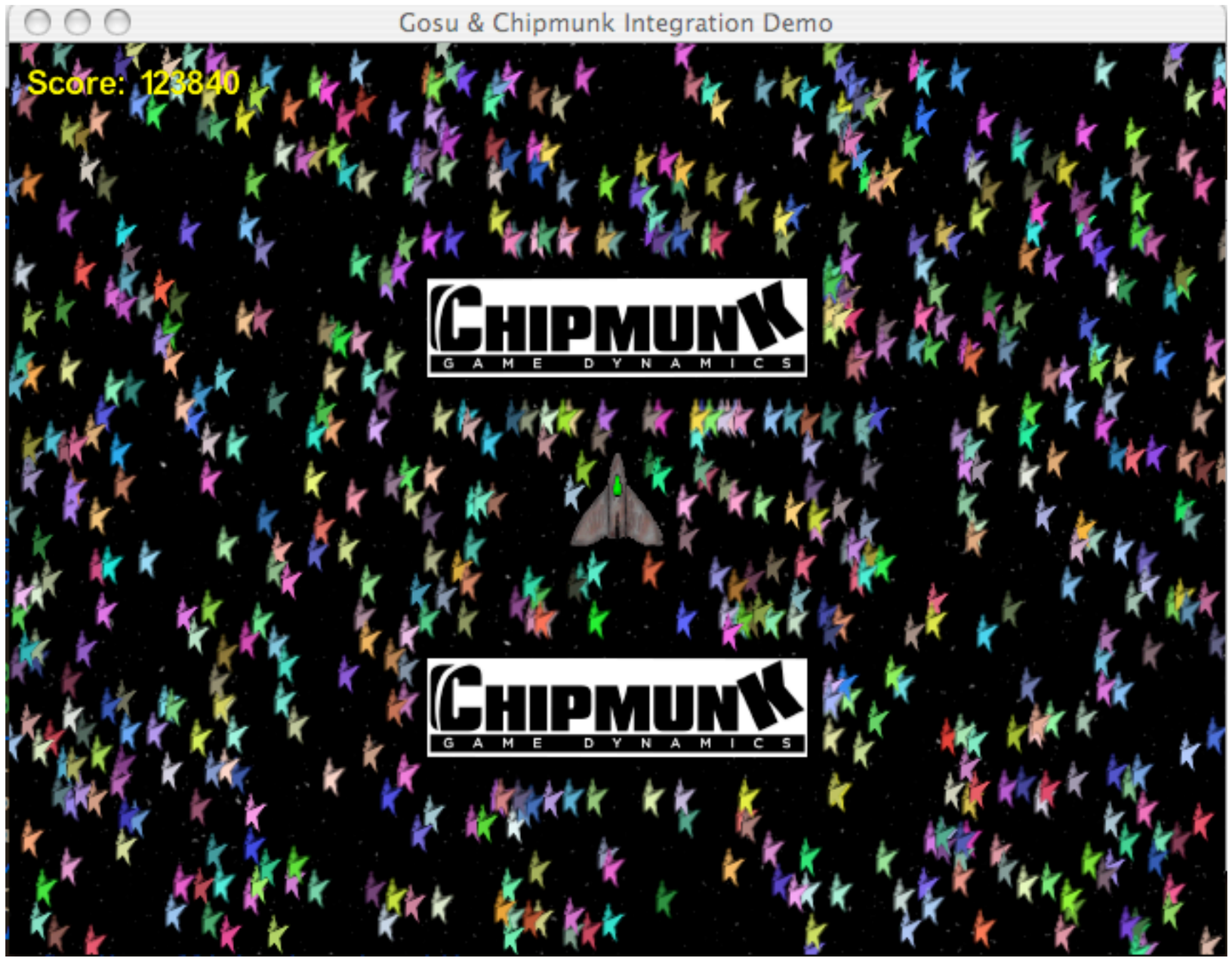
The star's body is mapped to a Chipmunk Circle. So a collision will register if another entity moves between the points of the star without actually touching the star. It's possible to assign multiple shapes to a body. Assigning a triangle to each point and a polygon to the middle of the star would make for a more precise game.

There are numerous changes to the internals of the original Gosu tutorial in the Chipmunk-enhanced version. But when you run the application, it looks almost exactly like the original version.

I think the most magical thing about Chipmunk is watching the stylized Sprites move in a natural way. So I modified Dirk's tutorial by commenting out the code that removes the stars when the spaceship steers into them. That way you could see what happens when the spaceship hits the Chipmunk-enhanced stars.

and

CHIPMUNK
GAME DYNAMICS



Based on a Tutorial By Dirk Johnson

I also made the star field more dense and added a couple of Chipmunk logos.

This is the sort of thing that can happen when you mix Gosu aand Chipmunk.

NOTE: THIS IS A MOVIE, NOT A STILL SCREEN SHOT

and

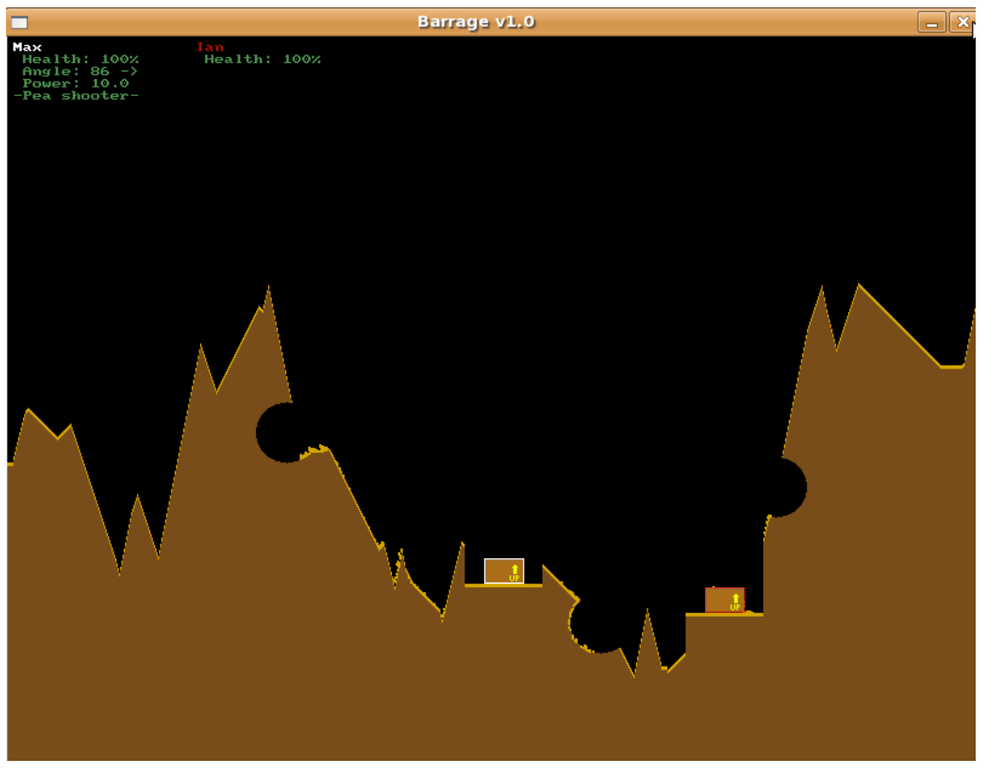
Sample Packaged with Gosu

Gosu was recently integrated with RMagick after users requested the ability to dynamically modify the landscape during a game. This sample app that is packaged with Gosu shows some of the special effects that were not possible without RMagick.

RMagick is the Ruby bindings for the ImageMagick libraries, which can be used to transform or combine images and apply dozens of special effects, like mirroring, flipping and even emulating a watermark or Polaroid instant picture.

and


```
def remove_dirt point, radius
  $playfield_buffer.filled_circle([point[0],
     point[1]], radius, [0,0,0])
  $backbuffer.filled_circle([point[0],point[1]],
    radius, [0,0,0])
end
```



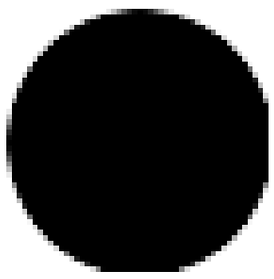
Sample Packaged with RUDL

“Barrage” by Brian Palmer, Pocket Martian Software

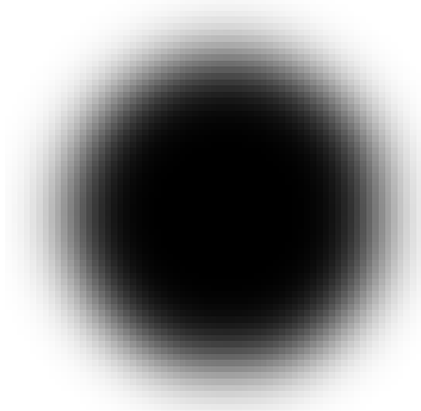
For comparison’s sake here are a couple of screen shots of a game packaged with RUDL. It is similar in that it also involves blasting holes in the landscape. It features tanks instead of toy soldiers. The little square boxes are the tanks. The remove_dirt method makes it look like dirt was displaced by rendering a black circle where the projectile dropped. This only works with a black background.

and


```
if not @crater_image then
  @crater_image = Magick::Image.new(50, 50)
    {self.background_color = 'none'}
  gc = Magick::Draw.new
  gc.fill('black').circle(25, 25, 25, 0)
  gc.draw(@crater_image)
  @crater_shadow = @crater_image.shadow(0,0,5,1)
end
```



@crater_image



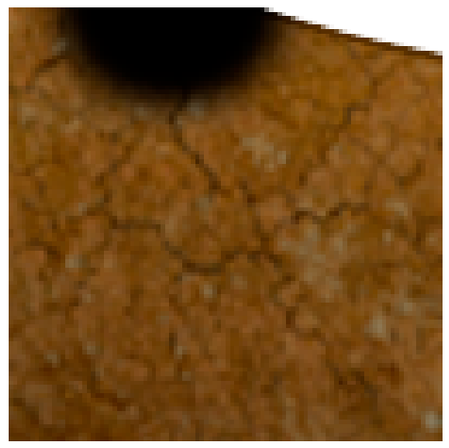
@crater_shadow

With an ImageMagick-based implementation there are no such restrictions. RMAgick integration opens up so many possbilities.

This slide and the next show the code that makes it look like craters with charred rims are left in the earth. In this code block “circle” and “fill” create a black circle, and “shadow” creates a blurred version of the circle.

and


```
@rmagick_images[index].composite!(@crater_shadow,
  center_x - 35, center_y - 35,
  Magick::AtopCompositeOp)
```



```
@rmagick_images[index].composite!(@crater_image,
  center_x - 25, center_y - 25,
  Magick::DstOutCompositeOp)
```

```
@gosu_images[index] = Gosu::Image.new(@window,
 @rmagick_images[index], true)
```

When one of the soldiers shoots the star or the ground, the app loops through the images tiles that comprise the background and determine which ones were near the blast.

It then uses the RMagick composite method, passing it the “Atop” flag to place the shadow on the tile image.

Next it uses the DstOutcompositeOp flag when it adds the smaller smoother crater image. This flag indcates that a space the shape of the image should be erased.

The last line of code generates a Gosu image based on the composite rmagick image, just as a Gosu image can be based on a PNG file.

and

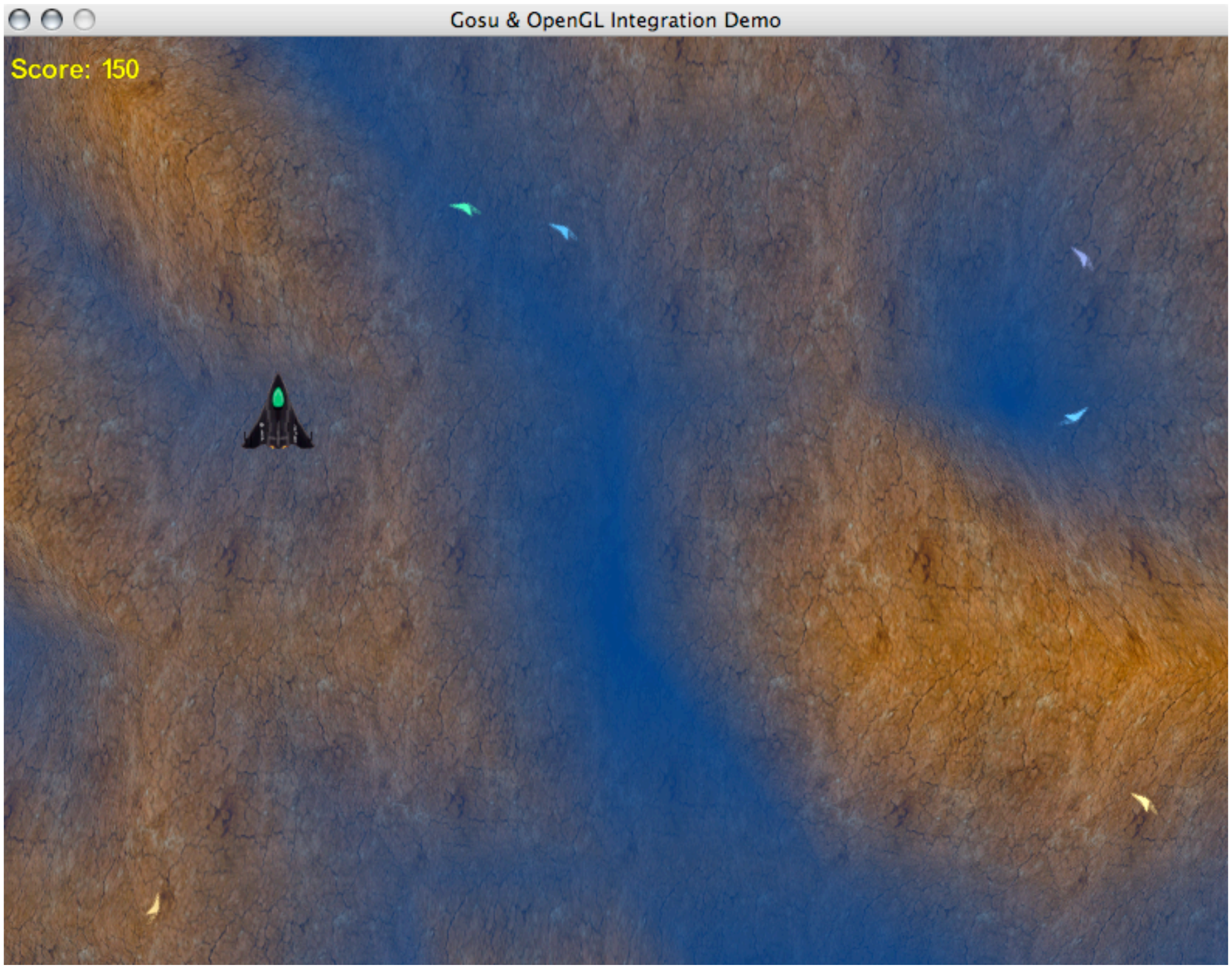

```
def draw
  gl do
    glClearColor(0.0, 0.2, 0.5, 1.0)
    glClearDepth(0)
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    @gl_background.exec_gl
  end
  ...
end
```

The very latest release features another variation on the Tutorial.

Gosu uses OpenGL for rendering behind the scenes. The version released just this past week enables developers to access the OpenGL API directly from the Window class, by putting the GL calls in a block and passing it to the “gl” method.

The new sample incorporates a 3D background. Most of the OpenGL API calls are encapsulated in
the exec_gl method which I will show you after we see the sample in action.

and



Sample Packaged with Gosu

The mountain range effect is created using the same texture over and over with a random height.

and


```
def exec_gl
  info = @image.gl_tex_info
  ...
    0.upto(POINTS_Y - 2) do |y|
      0.upto(POINTS_X - 2) do |x|
        glBegin(GL_TRIANGLE_STRIP)
          z = @height_map[y][x]
          glColor4d(1, 1, 1, z)
          glTexCoord2d(info.left, info.top)
          glVertex3d(-0.5 + (x - 0.0) / (POINTS_X-1),
            -0.5 + (y - offs_y - 0.0) / (POINTS_Y-2),
            z)
          ...
        glEnd
      end
    end
end
```

Here is a fragment of the exec_gl code.

The gl_tex_info method gives the Gosu application code access to the bound gl texture. The background is composed of multiple triangles, which have vertexes in common.

Instead of making a separate set of GLVertex calls for each surface, which would result in specifying the same vertex mor than once, the GL_TRIANGLE_STRIP OpenGL API specifier is used. With GL TRIANGLE STRIP you declare 3 or more vertexes and OpenGL takes care of connecting them all with Triangle surfaces. This slides shows just one of the 4 vertexes defined in the application. The scrolling background is composed of multiple GL_TRINAGLE_STRIPS.

Creator and Lead Developer:
Jason Roelofs

http://ogrerb.rubyforge.org/index.html

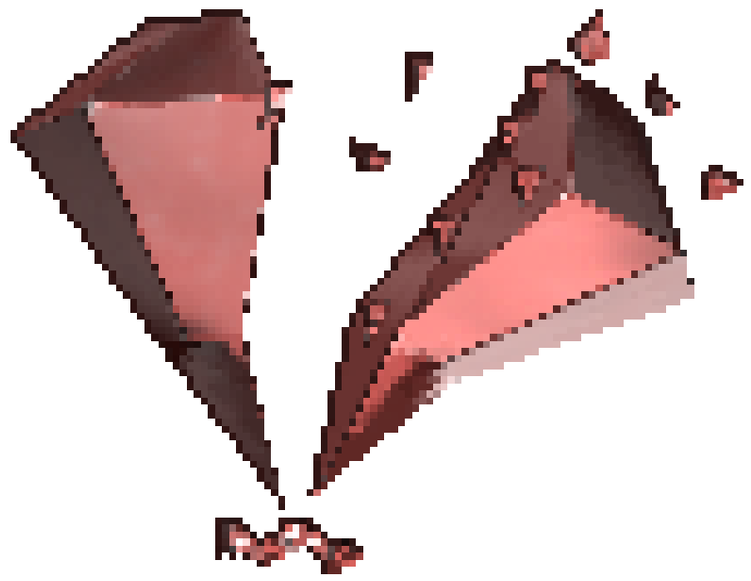
Given what it takes to hand code some 3D background effects using OpenGL, I'm sure you can imagine why people use 3D modeling software for more complex characters and scenes. Now we're going to look at 2 Ruby projects that leverage full-blown 3D models: Ogre.rb, a 3D graphics rendering engine, and ShatteredRuby, a 3D game developement framework that is built on top of Ogre.rb.

Ogre.rb is a Ruby wrapper around OGRE, which stands for Object-Oriented Graphics Rendering Engine. It is used for combining 3D models, textures and other kinds of graphical content into a scene. There are ways to incorporate special effects including wind, rain, smoke, photo-realistic explosions and sophisticated lighting techniques.

OGRE.RB Samples

This is a screenshot from one of the sample scenes packaged with Ogre.rb. It's actually a Ruby port of one of the demos packaged with OGRE. The green Ogre head is an example of a 3D model. Using arrow keys or the mouse, you can navigate around the screen and view the Ogre head from the side or from the back. The front is not just a facade. It's fully rendered in fine detail 360 degrees around.

OGRE is written in C++., and SWIG was used to help generate the C++ extension for Ruby. The Ogre.rb developers would like to incorporate Ruby style to the greatest extent possible. They expect it to be a challenge because the C++ best practices embraced by the OGRE team are hard to translate into idiomatic Ruby without making the API unrecognizable to experienced OGRE users.



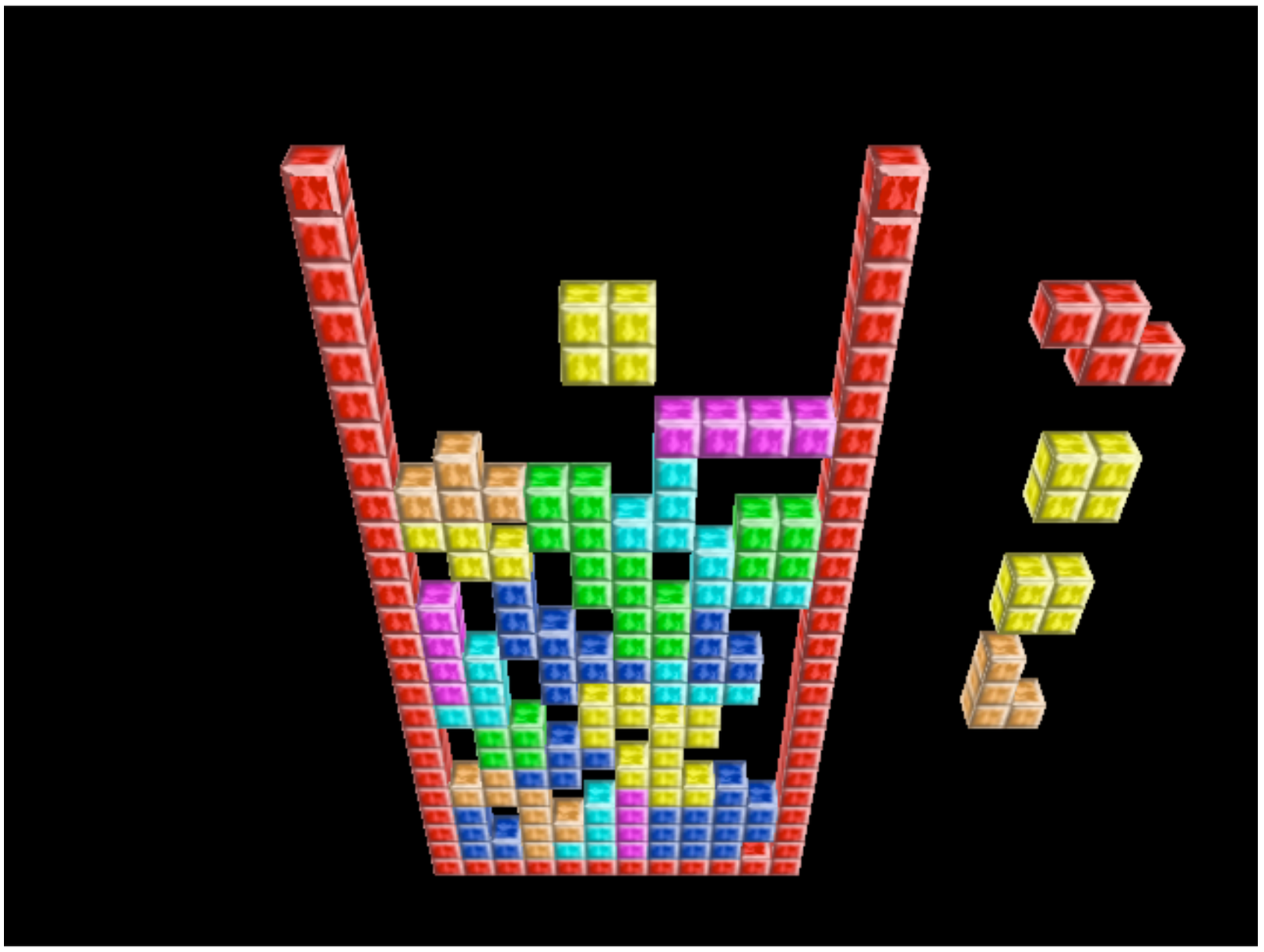
Shattered Ruby

Co-Creators and Co-Lead Developers:
Martyn Garcia and Mikkel Garcia

http://groups.google.com/group/shatteredruby

ShatteredRuby is a 3D game development framework inspired by Ruby on Rails and built on top of OGRE.

Shattered Ruby

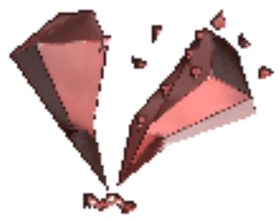


Shattered Tetris Tutorial

Here’s a version of tetris created with ShatteredRuby.

Shattered Ruby

Martyn Garcia on Shattered Ruby vs. Rails



"...while Rails has a simple text output view, Shattered has a very complex 3d rendering. And while Shattered has simple game logic, Rails has a very complex database backend.

Our focus is swapped.

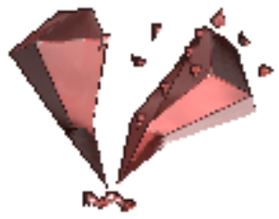
The way Rails gains a ton of power is by making assumptions on the model. ..[o]ur view is the largest part..."

It amazes me the extent to which Martyn and Mikkel find ways to talk about every aspect of 3D game development in terms of Ruby on Rails.

For example, here are some quotes from a post to the ShatteredRuby forum in which Martyn manages to liken Shattered to Rails -- even as he's pointing out a way in which Rails and Shattered are not only different, but diametrically opposed.

He points out that the Model in Rails is more complex than the typical Shattered Model,
while the View in Shattered -- a 3D view -- is way more complex than the View in a Rails app.

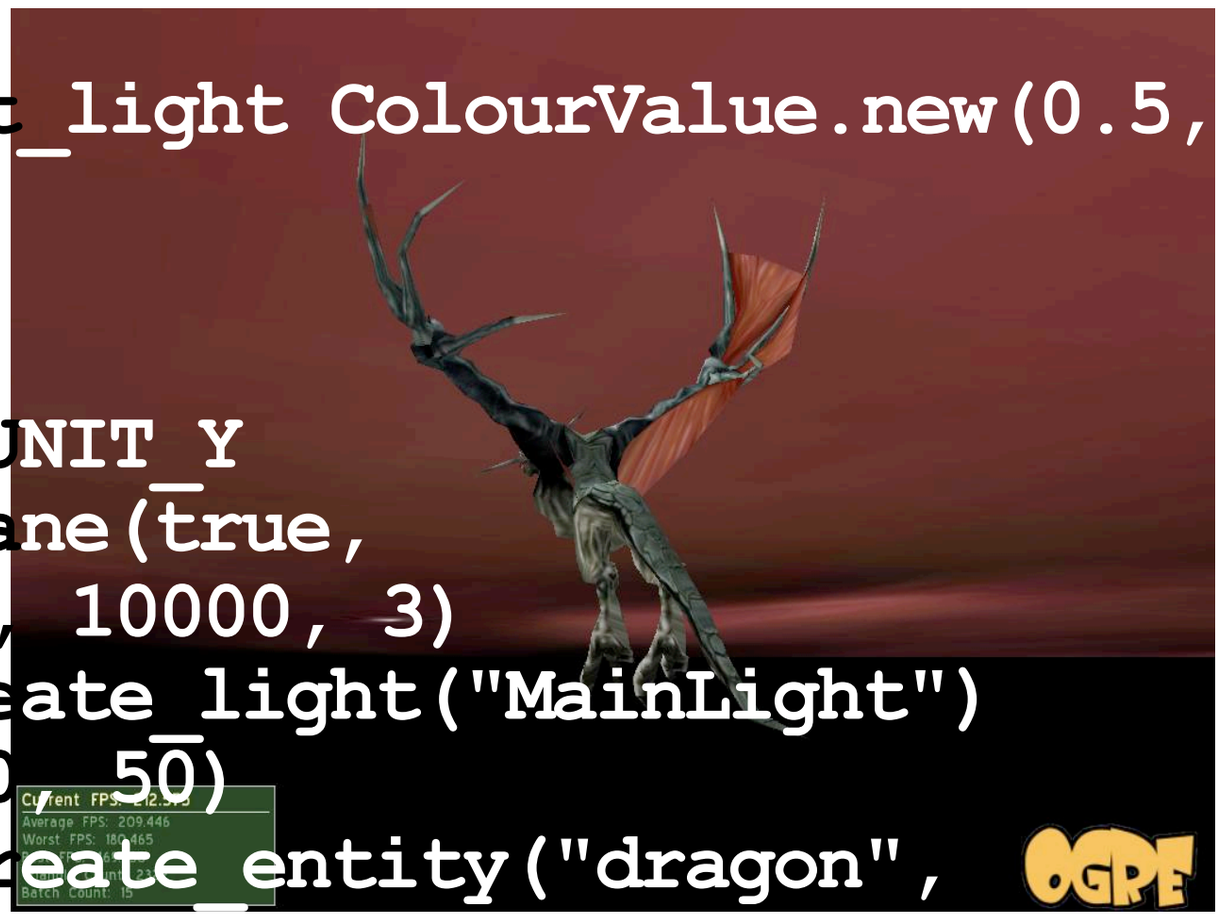
He concludes that Rails gets a lot of mileage out of being able to make assumptions about the Model, while Shattered can do the same with its Views.



ShatteredRuby

From the Ogre.rb Samples

```
class SkyPlaneApplication < Application
  def create_scene
    scene_manager.set_ambient_light ColourValue.new(0.5,
       0.5, 0.5)
    plane = Plane.new
    plane.d = 5000
    plane.normal = -Vector3.UNIT_Y
    scene_manager.set_sky_plane(true,
       plane,"SpaceSkyPlane", 10000, 3)
    light = scene_manager.create_light("MainLight")
    light.set_position(20, 80, 50)
    dragon = scene_manager.create_entity("dragon",
       "dragon.mesh")
    scene_manager.root_scene_node.attach_object(dragon)
  end
end
```



Shattered can reduce the amount of code needed to describe a scene considerably by making assumptions about where the media files can be found in the application directory structure and what they are called, about how objects created from a View are likely to be grouped, and about the relationship between the View and the Model.

Here's a code snippet from one of the sample apps that comes with OGRE.rb.

ShatteredRuby

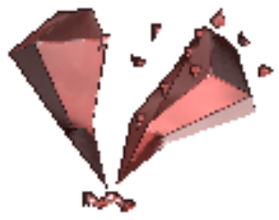
From the Ogre.rb Samples

```
class SkyPlaneApplication < Application
  def create_scene
    scene_manager.set_ambient_light ColourValue.new(0.5,
        0.5, 0.5)
    plane.normal = -Vector3.UNIT_Y
    scene_manager.set_sky_plane(true,
       plane, "SpaceSkyPlane", 10000, 3)
    lig                                    inLight")
    lig
    dra                                    dragon",
    sce                                    bject(dragon)
  end
end
```

Shattering the Code....

```
class SkyPlane
  light “Mainlight”,
        :position => (20,80,50)
end

class Dragon
end
```

The bottom block shows how the same scene could be expressed in Shattered.



Shattered Ruby

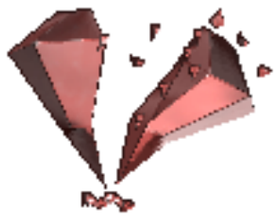
Symbols to Vector

```
:x.to_v             #v(1,0,0)
:y.to_v             #v(0,1,0)
:z.to_v             #v(0,0,1)
:up.to_v            #v(0,1,0)
:down.to_v          #v(0,-1,0)
:left.to_v          #v(-1,0,0)
:right.to_v         #v(1,0,0)
:forward.to_v       #v(0,0,1)
:backward.to_v      #v(0,0,-1)
:zero.to_v          #v(0,0,0)
```

But Shattered is not only about reducing code size -- it's also about having fun with the code you do write and being more expressive.

Vectors are most commonly created by passing the 3 coordinates to v.

I like the way Shattered provides a core extension for Symbol to add "to_v" support for symbols that describe changing direction, like forward and backward.



Shattered Ruby Timers

```
class Bomb < ...
   timer :in =>  30.seconds,
         :action => :blow_up
   timer :every => 1.second,
         :action => :shorten_fuse

   def blow_up
   ...
   end

   def shorten_fuse
   ...
   end
end
```

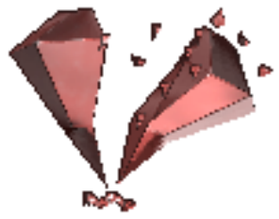
from the Shattered Wiki: http://wiki.shatteredruby.com/index.php?title=Timers

Timers are what makes Shattered tick. Shattered makes it easy to set up Timers. I went ahead and took these examples from the Timers page Shattered Wiki because I couldn't think of a better example of a timer demo class than a bomb that needs its fuse shortened every second until it blows up in 30 seconds.

ShatteredRuby

```
> shatter space_race
```

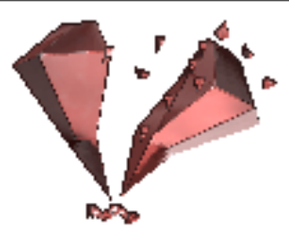
Let’s back up now and look at how a typical Shattered app is structured. You begin building a game by typing shatter and the name of your game.



Shattered Ruby

```
> shatter space_race

   create  app/models
   create  app/views
   create  app/media/common/programs
   create  app/media/common/templates
   create  doc
   create  config
   create  log
   create  script
   create  test/unit
   create  vendor
   create  vendor/plugins
   create  Rakefile
   create  README
   create  config/ogre_plugins.windows.cfg
   create  config/ogre.cfg
   create  config/boot.rb
   create  config/environment.rb
   create  test/test_helper.rb
   create  script/generate
   create  script/runner
   create  script/console
   ...
```

Shattered generates this familiar-looking directory structure.



ShatteredRuby

Model\View\Actor Architecture

```
space_race>script/generate actor flying_carpet
      exists  app/models/
      exists  test/unit/
      create  app/models/flying_carpet.rb
      create  test/unit/flying_carpet_test.rb
      exists  app/views/
      create  app/media/flying_carpet
      create  app/views/flying_carpet_view.rb
      create  app/media/flying_carpet/flying_carpet.mesh
      create  app/media/flying_carpet/flying_carpet.png
```

Actor

Model

View

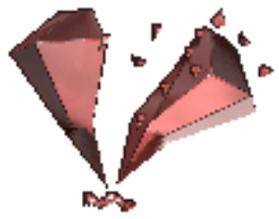
Model

Contains game logic and handles user input events.
When a method is called on the Model, and a method with the same name exists on the View -- the View's method is automatically invoked.

View

The View responds to methods on the Model.

Actors are made up of a Model and a View. The Model contains the game logic and the user input handlers.
Methods on the Model that are also defined on the View will get invoked.



ShatteredRuby

State Management

```
space_race>script/generate state menu
      create  app/states/
      create  app/states/menu_state.rb

space_race>script/generate state game
      exists  app/states/
      create  app/states/game_state.rb
```

State

Represents routes through the application

Provides a top level where actors can interact

A game should have at least one State. You can generate a State using script/generate.

ShatteredRuby

OGRE Materials Format

```
// This is a comment
material walls/funkywall1
{
    // first, preferred technique
    technique
    {
        // first pass
        pass
        {
            ambient 0.5 0.5 0.5
            diffuse 1.0 1.0 1.0

            // Texture unit 0
            texture_unit
            {
                texture wibbly.jpg
                scroll_anim 0.1 0.0
                wave_xform scale sine 0.0 0.7 0.0 1.0
            }
            // Texture unit 1 (this is a multitexture pass)
            texture_unit
            {
                texture wobbly.png
                rotate_anim 0.25
                colour_op add
            }
        }
    }

    // Second technique, can be used as a fallback or LOD level
    technique
    {
        // .. and so on
    }

}
```

(from the OGRE manual:
http://www.ogre3d.org/docs/manual/manual_14.html)

Ogre materials scripts can be used to control textures, light settings and positioning for 3D models. This is the format for material scripts, as specified in the OGRE manuals.

Shattered Ruby Rmaterials

OGRE Materials

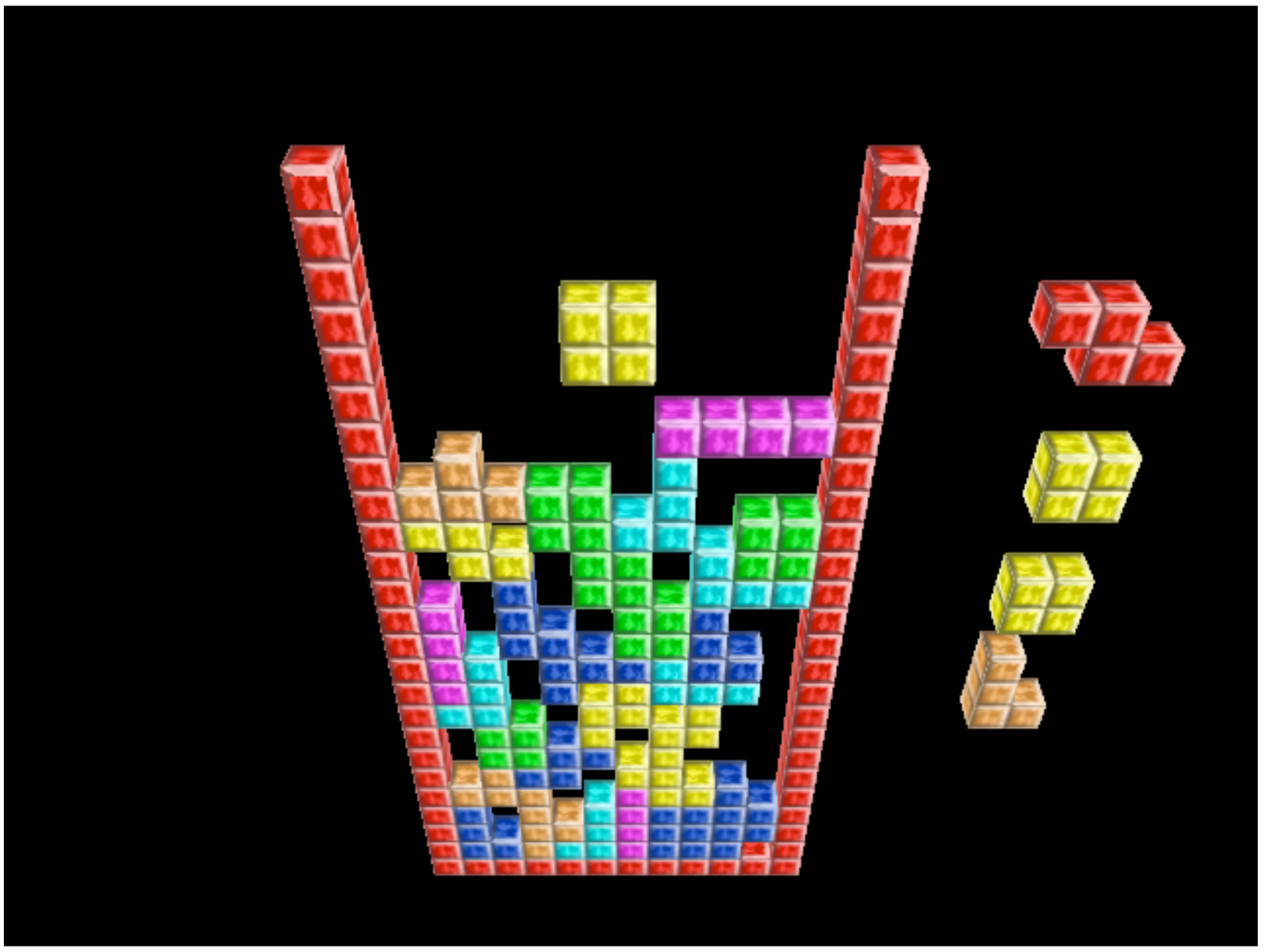
```
// This is a comment
material walls/funkywall1
{
    // first, preferred technique
    technique
    {
        // first pass
        pass
        {
            ambient 0.5 0.5 0.5
            diffuse 1.0 1.0 1.0

            // Texture unit 0
            texture_unit
            {
                texture wibbly.jpg
                scroll_anim 0.1 0.0
                wave_xform scale
            }
            // Texture unit 1 (this
            texture_unit
            {
                texture wobbly.png
                rotate_anim 0.25
                colour_op add
            }
        }
    }

    // Second technique, can b
    technique
    {
        // .. and so on
    }
```

```
material < %= name % >
{
    technique
    {
        pass
        {
            lighting off
            texture_unit
            {
                texture < %= texture % >
            }
        }
    }
}
```

```
material :flying_carpet, :template => 'basic',
         :texture => 'tapestry.png'
```

Shattered supplies a mechanism for using erb to generate scripts with the Ogre materials format.

Shattered Ruby



Shattered Tetris Tutorial

Shortly after the latest version of tetris tutorial was posted on the Shattered Wiki last year, Martyn and Mikkel began a major restructuring effort.

One of the major goals was to get rid of the Controller in favor of the application structure we look at a few slides back. It used to be that a Actor was paired with an Model\View\Controller trio. The game logic lived in the Model, the View was told when and what to display, and the Controller handled user requests and was supposed to tap the Model and View as needed. Mikkel and Martyn recognized that it was too easy to put certain kinds of logic in both the Controller and the Model.

Shattered Ruby

That's right, there are no more "What, Shattered doesn't support per vertex bi-quadruple render transmutation?! -- that's outragous!". Anything supported in Ogre will be supported in Shattered!

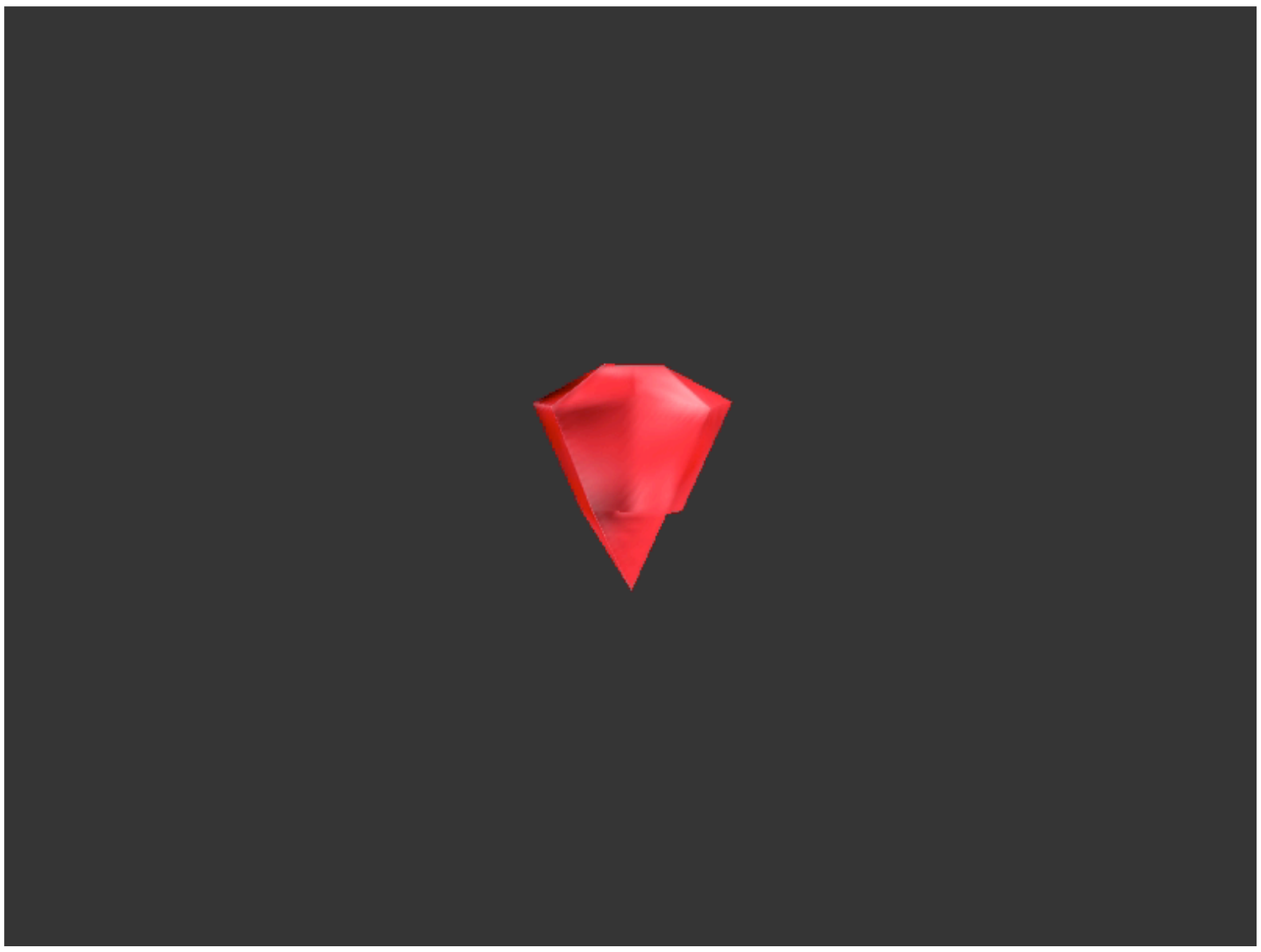
-- Mikkel Garcia

The other main goal was to make the everything supported by Ogre.rb available to Shattered developers. When the framework was first written, only wrappers for the subset of Ogre features that were needed for Shattered's game SDL were packaged with Shattered. On the Shattered blog, Mikkel explains this decision in terms fo Ruby on Rails. He explains that Shattered and Ogre.rb are analogous to ActiveRecord and Rails. Just as Rails developers have the option of using straight SQL when there's no ActiveRecord support for something they need to do, Shattered developers can use straight Ogre.rb calls when Shattered doesn't handle something out of the box.

So, I can't demo tetris for you now, eventually, the tutorial on the Shattered Wiki will be upated and you will be able to download it for yourself.

So what can you do with Shattered now. with the current downloadable version of Shattred and the somewhat dated tutorials on the Shattered Wiki?

Shattered Ruby Tutorial



http://wiki.shatteredruby.com/

I think the coding enviroment and style is amazing. This is pretty much all the application code you need.

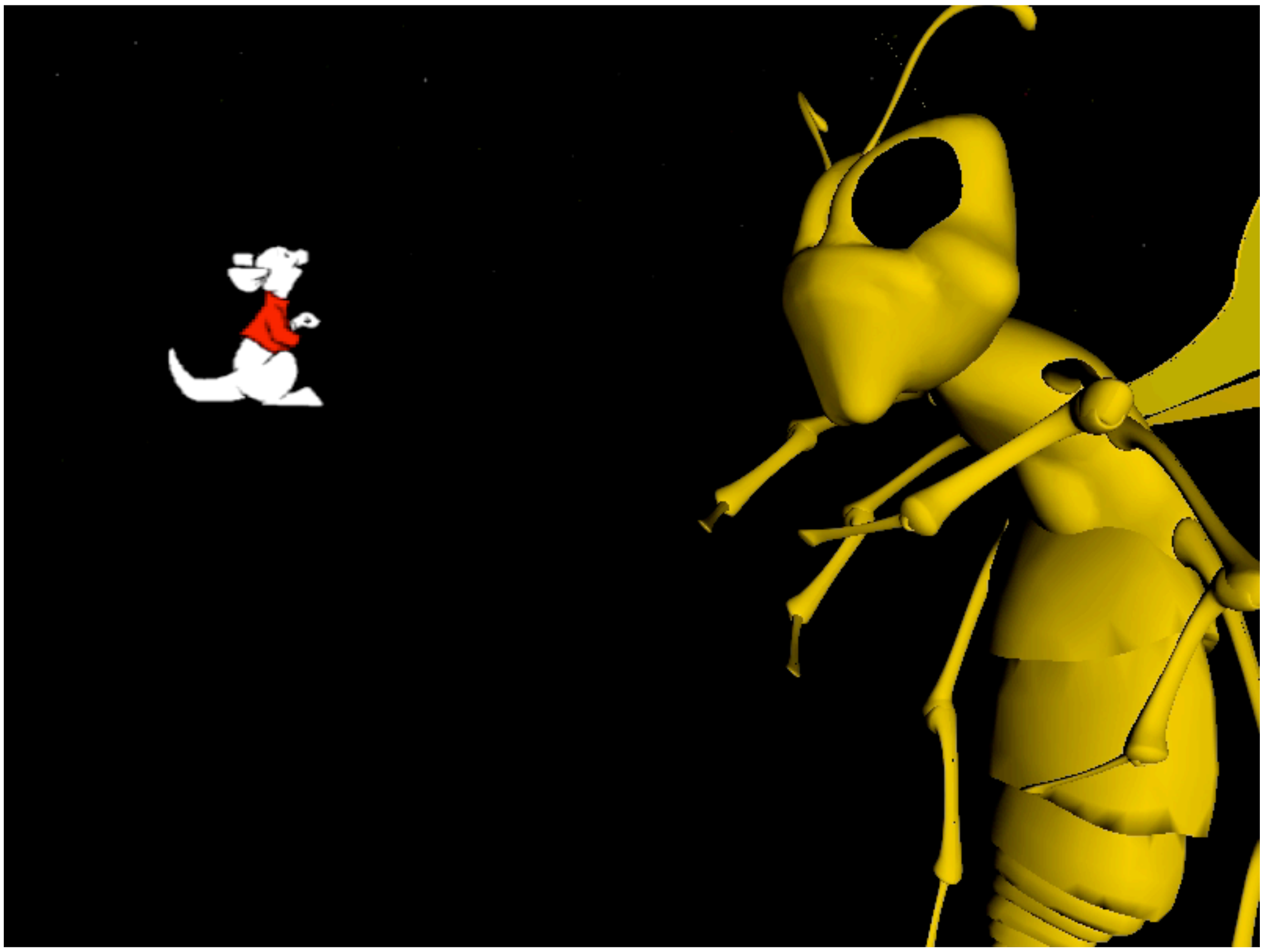
But running the app isn't particularly satisfying. We saw this sort of thing with the the OpenGL API.

3D is most satisfying when working with more assymetical models. You can create these with modeling tools. There's a great open source one called Blender. There are also hundreds of 3D models on the web that you can download for free.

When I rotate these it gives me the sensation of actually reaching into my laptop and spiining the objects around with my own hands.

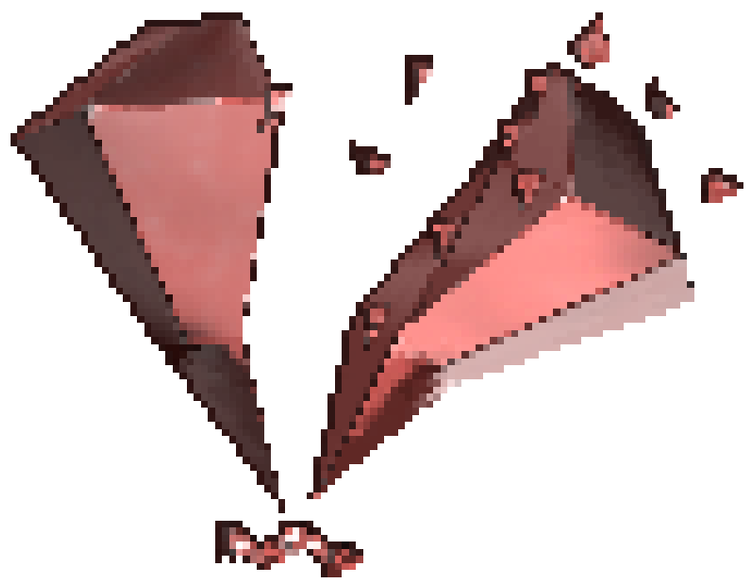
So I'm just going to show you a sample app I made wit the current version of Shattered. Maybe it will give you some ideas. It shows you that you can move models and detect their position. And there's an incredible amount you can do wiht jyst that.

Shattered Ruby



Here’s the end result of the “first moves” tutorial on the Wiki.

It pretty much works with the current codebase. You can check the shattered google group for minor issues people ahve run into.



Shattered Ruby

Co-Creators and Co-Lead Developers:
Martyn Garcia and Mikkel Garcia

http://groups.google.com/group/shatteredruby

The best way to find out when the tutorial has been updated and when the next release is out is to join the google group.

Creator and Lead Developers: David Koontz

http://rubyforge.org/projects/railgun/

Like shattered ruby, railgun is a 3D framework that aims to provide a rails-like development experience. It's JRuby-based and wraps JMonkeyEngine, a 3D game framework written in Java.
It is in it's beginning stages, but I think it will be intresting project to watch.

It's creator David Koontz is here and you can ask him to do a demo maybe. He has also created a rails-like framework for creating Swing GUIs.

High Art

on top of

LOW-LEVEL APIS

Building Games with Ruby

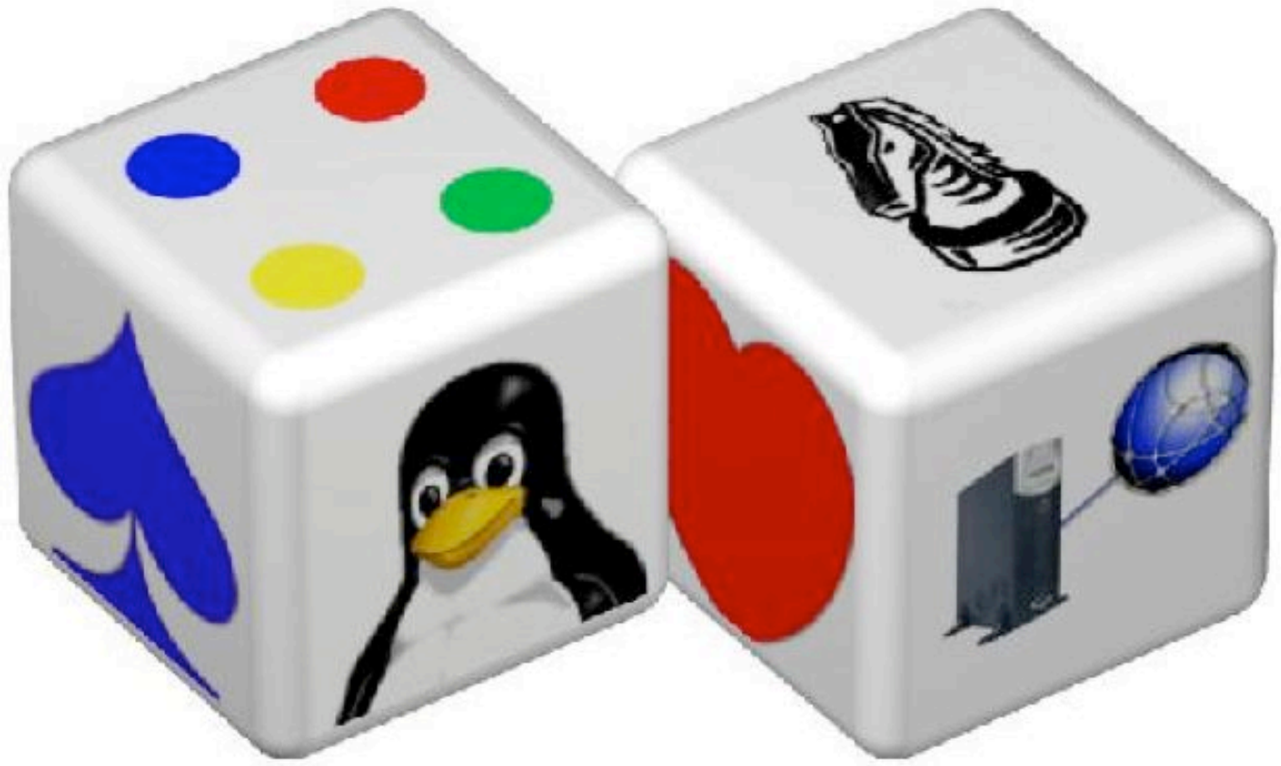
Agenda

1. Buiding 2D Games with Ruby
2. Building 3D Games with Ruby
3. Is Ruby a legitimate player in the gaming space?

Well ‘d like to start addressing that topic by telling you about something that is both indicative of
Ruby’s rising profile in the gaming space and something that has the potential to further improve Ruby’s standing as a game programming language.

And that is that the GGZ Gaming Zone has bolstered it’s Ruby support.

The GGZ Gaming Zone Project



Founders: Brent M. Hendricks and Rich Gade

Developed & Maintained by
an International Team

http://www.ggzgamingzone.org/

GGZ is a recursive acronym, like GNU. Actually, when GGZ was founded around 8 years ago, it was initially called Gnu Gaming Zone, or GGZ. The founders anticipated becoming part of the GNU project, but when they later decided against that, they decided to go with the name GGZ Gaming Zone.

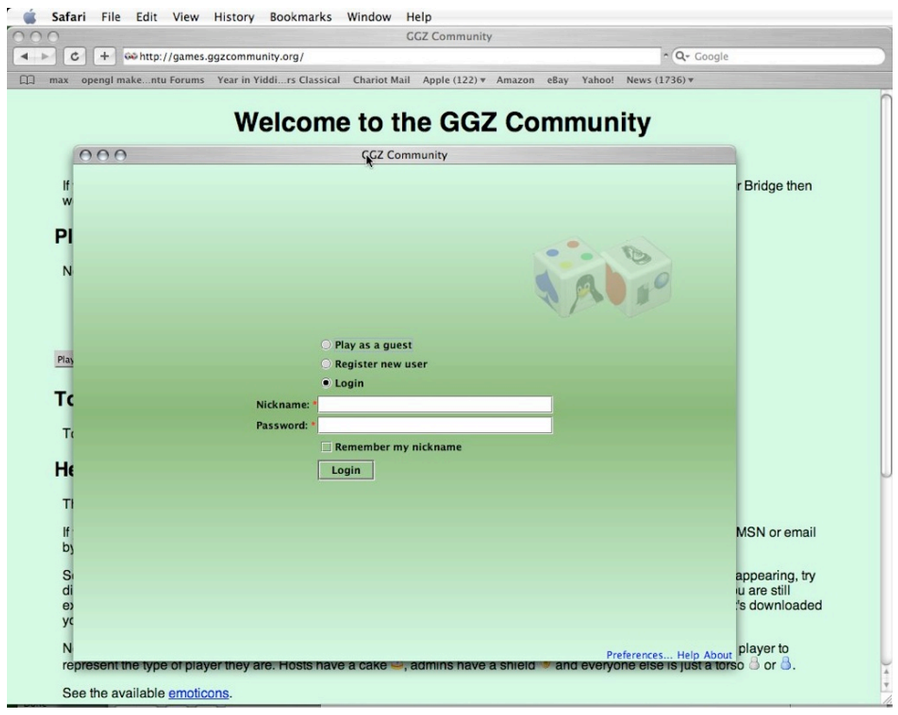
The GGZ Gaming Zone project has a lot of useful resources for game developers, including libraries that facilitate separating your game into client and server components, libraries and protocols for deploying your game over a network, and libraries that provide services ranging from sign in to reserving seats to saving high scores.

The GGZ source repository also contains dozens of games, and a Web portal designed for gaming communities -- complete with RESTful APIs for accessing information like tournament schedules and player rankings.

The portal software is running on the GGZ community site, where you can also find tournament schedules. Spades is particularly popular. There are Spades tournaments on a regular basis.

GGZ Gaming Zone

http://games.ggzcommunity.org/



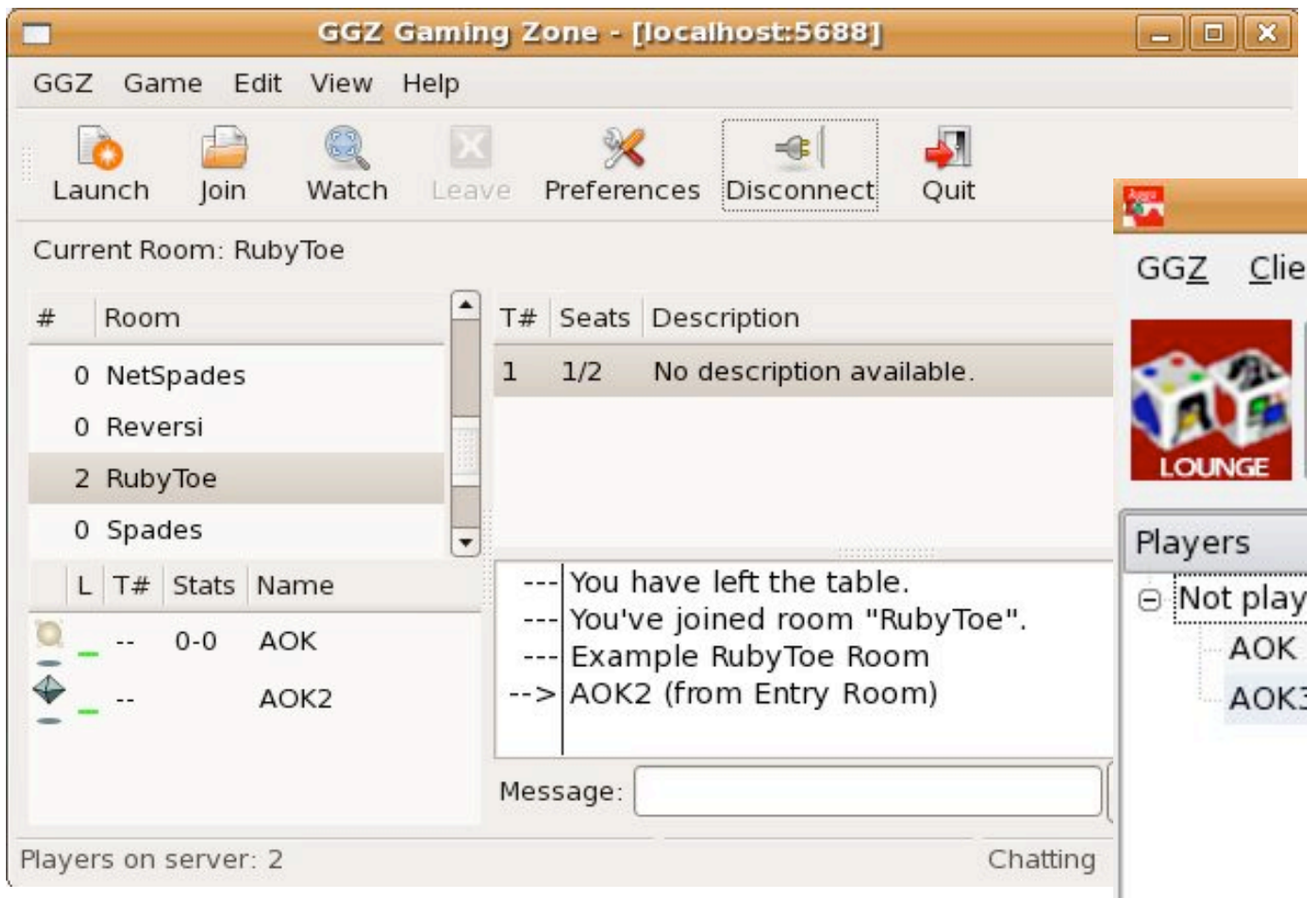
http://live.ggzgamingzone.org:5688/
localhost:5688/

The top half of this screen shows the logon screen that pops up when you go the ggz community site and hit the “Play Now” button. You don’t even need an account to play the games that are installed there. You can just logon as guest.

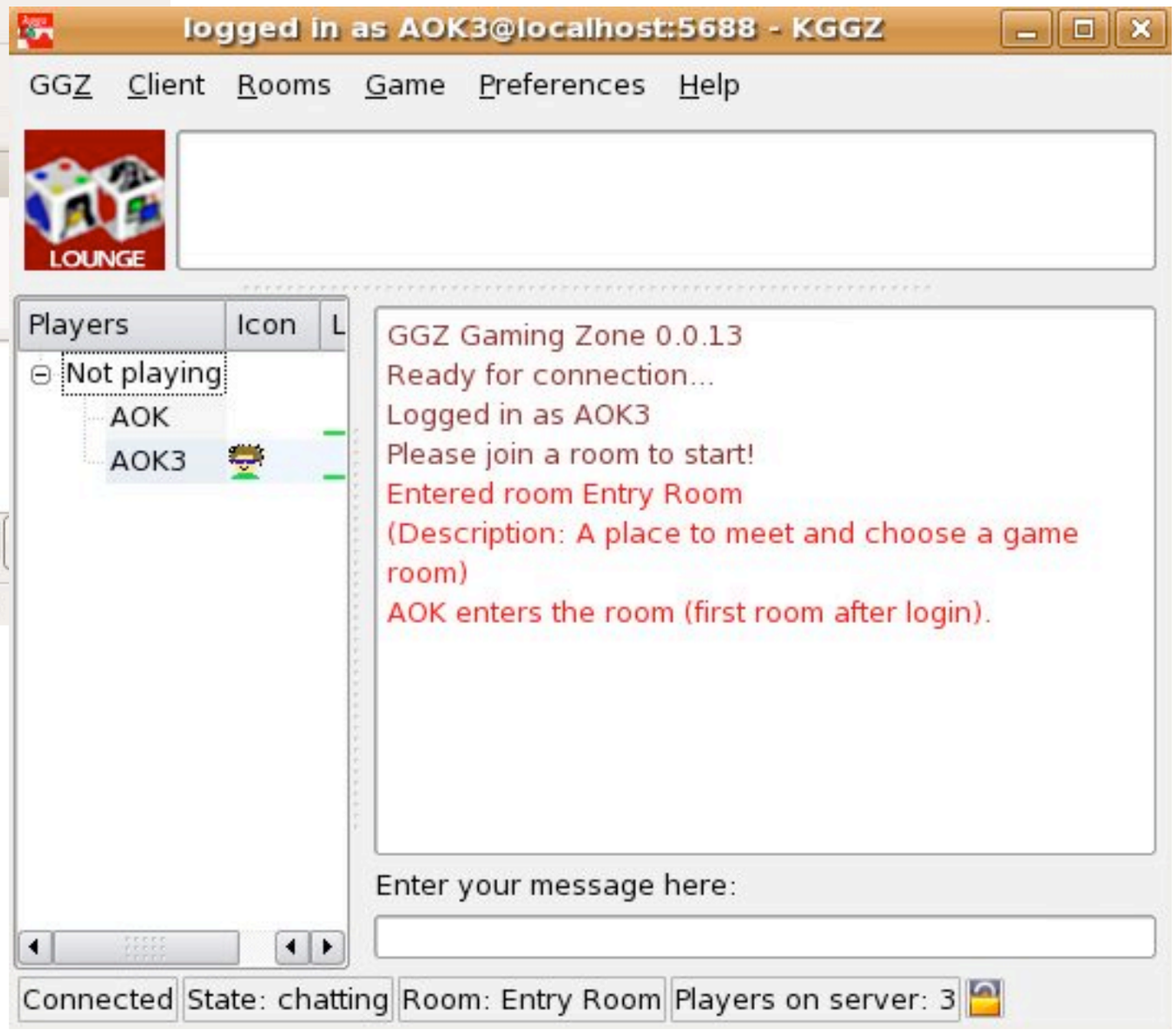
The bottom half of the split screen shows the logon screen you’ll see if you install GGZ software on your computer, and run the core launching client. In this screen you can choose the ggz community portal server, you can choose the ggz server on your own local machine, the ggz server on your friend’s machine ... or any number of other public GGZ servers running all over the world.

The GGZ core launching client shown here is the GTK+ client. There is also a KDE client -- and until recently there was a GNOME client. The author of GnomeChess is working on a new GNOME launching client for GGZ.

GGZ Gaming Zone



GGZ GTK+ Core Client



GGZ KDE Core Client

The core clients packaged with GGZ are basically chat clients that can launch games.

The core client lists games that are available for launching and provides information about the status of any virtual tables (think “bridge table”, not database table) where games are in progress or where people are waiting for additional players to show up. When you launch a multiplayer game, you are prompted to indicate which seats should be open to anyone, which seats should be reserved for particular players, and which should be assigned to the computer. You can join games as a player or a spectator.

GGZ Gaming Zone

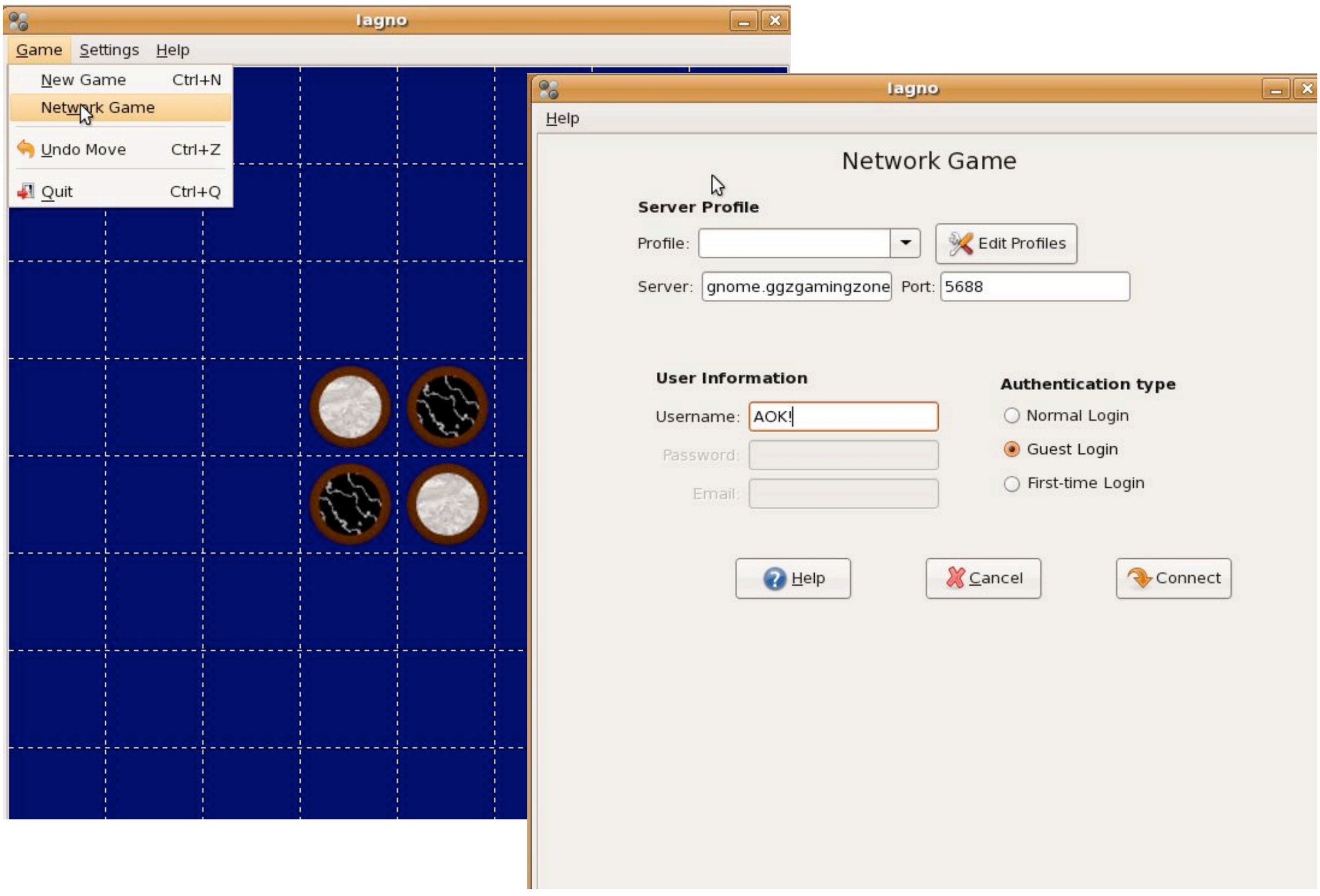
Here's a sampling of games packaged with GGZ.

GGZ game code is usually broken down into game server code and game client code.

GGZ offers more than one client for several of the games. Three very different TicTacToe clients are on the bottom here. If there are multiple front ends available, the core launching client will prompt you to pick one.

GGZ Gaming Zone

Iagno, GNOME Games



Here's an example of a Game that is GGZ-compliant, but is not packaged with GGZ.

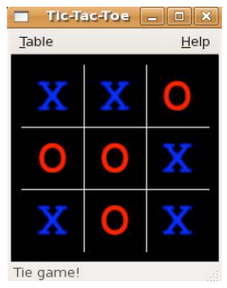
Iagno is part of the GNOME Games package. All of the GNOME games have GGZ Core launching clients built into them. This is the screen you'll see if you choose Network Game from the Game menu shown. You can choose a remote game server and logon to play Iango.

There are other games that are GGZ compliant, but not packaged with GGZ, that are hosted on GGZ's server.

GGZ Gaming Zone

ggzcommgen

game client



net.c ⟷ **net.c**

Game Client/Game Server

Individual Protocol

TCP/IP or UDP/IP

game server (tictactoe)

ggzmod/game

ggzdmod/server

Game Client/Core Client

Binary Protocol

Local Connection

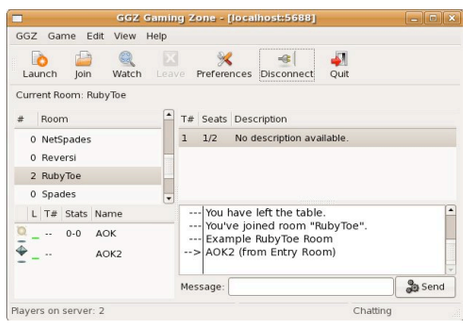
Game Server/GGZ Server

Binary Protocol

Local Connection

ggzmod/core

core client



ggzcore

Core Client/GGZ Server

XML Protocol

TCP/IP

ggzmod/core

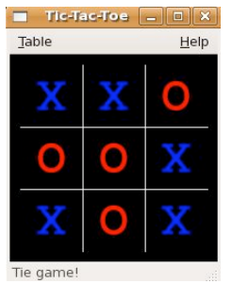
ggz server

This architecture that makes it easy for GGZ to support mulitple core clients, and multiple game clients and multiple game servers for the same game.

GGZ's publishes protocols for communication between clients and servers that define message types and describe the kinds of data that must accompany each message type.

GGZ Gaming Zone

game client



ggzmod/game

Game Client to Core Client

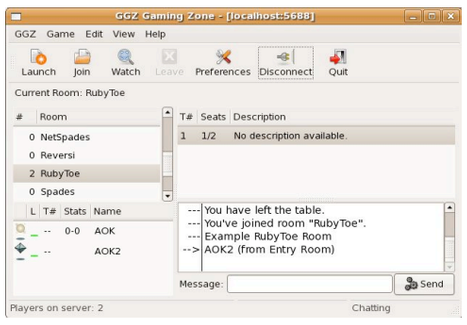
GAME_STATE(0), STAND(1), SIT(2), BOOT(3), BOT(4), OPEN(5), CHAT(6)

Core Client to Game Client

GAME_LAUNCH(0), GAME_SERVER(1), GAME_PLAYER (2), GAME_SEAT(3), GAME_SPECTATOR SEAT(4) GAME_CHAT(5), GAME_STATS(6)

ggzmod/core

core client

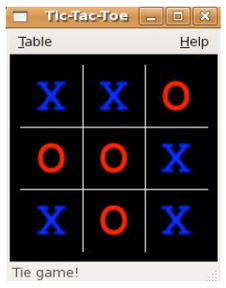


Here is a closer look at one of these protocols. These are the message types included in the communication protocol between the Game Client and the Core Launching Client.

Each message type is assigned a numeric id as part of the protocol. The ids are shown in white here.

GGZ Gaming Zone

game client



ggzmod/game

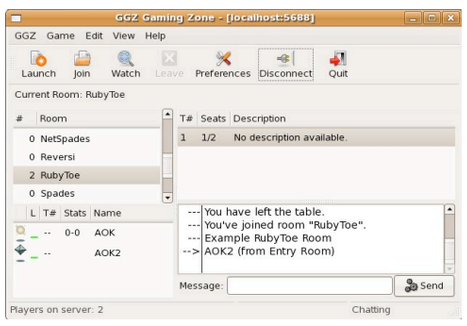
DATA	TYPE
Opcode	4
Spectator Seat#	integer
Spectator Name	string

Game Client

GAME_LAUNCH(0), GAME_SERVER(1),
GAME_PLAYER (2), GAME_SEAT(3),
GAME_SPECTATOR SEAT(4)
GAME_CHAT(5), GAME_STATS(6)

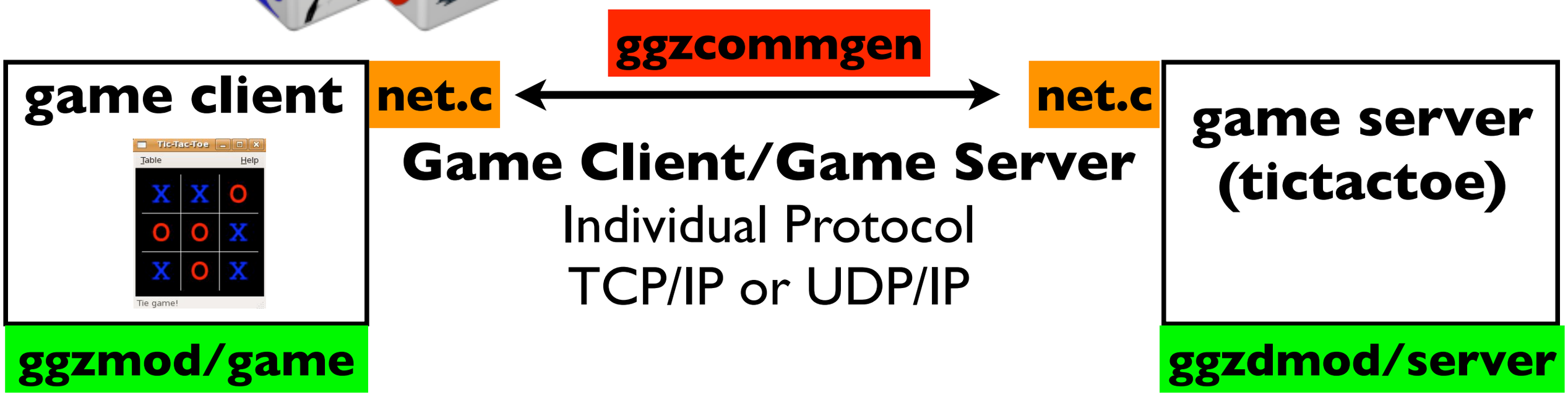
ggzmod/core

core client



GAME_SPECTATOR_SEAT is an example of a type of message a core client can send to a game client to indicate that a user wants to join the game as a spectator. The protocol specifies that the seat id -- an integer -- and the player name -- a String -- must be sent with a GAME_SPECTATOR_SEAT message.

GGZ Gaming Zone



Game Client/Core Client

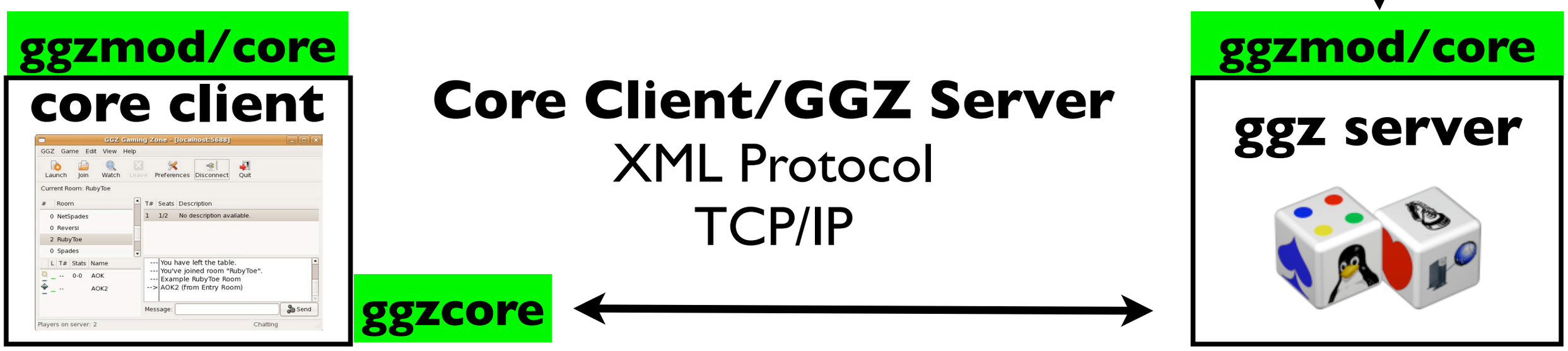
Binary Protocol

Local Connection

Game Server/GGZ Server

Binary Protocol

Local Connection



For everything other than the communication between a game client and a game server, GGZ provides utility libraries that encapsulate both the GGZ-specific protocols and the network protocols used to send the messages. These libraries are represented by the highlighted green text in the diagram.

Since each individual game has a different protocol for communication between the game client and the game server -- GGZ can't offer a pre-packaged library that will work for every game.

What it does offer, instead, is a flexible code generation utility called ggzcomgen. You define your game protocol in XML format and specify your network protocol as a command line argument, and the utility will generate a file that contains a method for each message type in your game protocol that uses the specified network protocol to send the appropriate data types over the network.

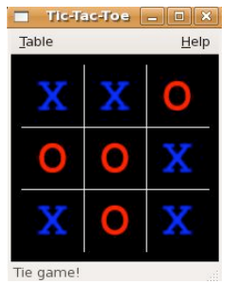
The generated networking code is represented by the filenames highlighted in orange. I called the generated files net.c in this diagram because "net.c" is the name of the file the utility generates for games written in C, and C is the primary development language for GGZ.

The code generator itself, ggzcommgen, is highlighted in red because it is written in Ruby. It is the only part of the GGZ infrastructure that is written in Ruby.

GGZ Gaming Zone

ggzcommgen

game client



net.c ⟷ **net.c**

Game Client/Game Server

Individual Protocol

TCP/IP or UDP/IP

game server (tictactoe)

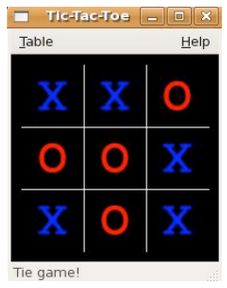
```
<ggzcomm engine="tictactoe" version="4">
  <definitions>
     <def name="msggameover" value="3"/>
  </definitions>
  <server>
   ...<event name="msggameover">
        <data name="winner" type="byte"/>
     </event>
  </server>
</ggzcomm>
```

Here's an example of an XML file that describes the tictactoe game client\game server protocol. You can see that the "game over" message is given a numeric id value of 3 -- and that a single byte value, called "winner" should be sent along with the message.

GGZ Gaming Zone

ggzcommgen

game client



net.c ⟷ **net.c**

Game Client/Game Server

Individual Protocol

TCP/IP or UDP/IP

game server (tictactoe)

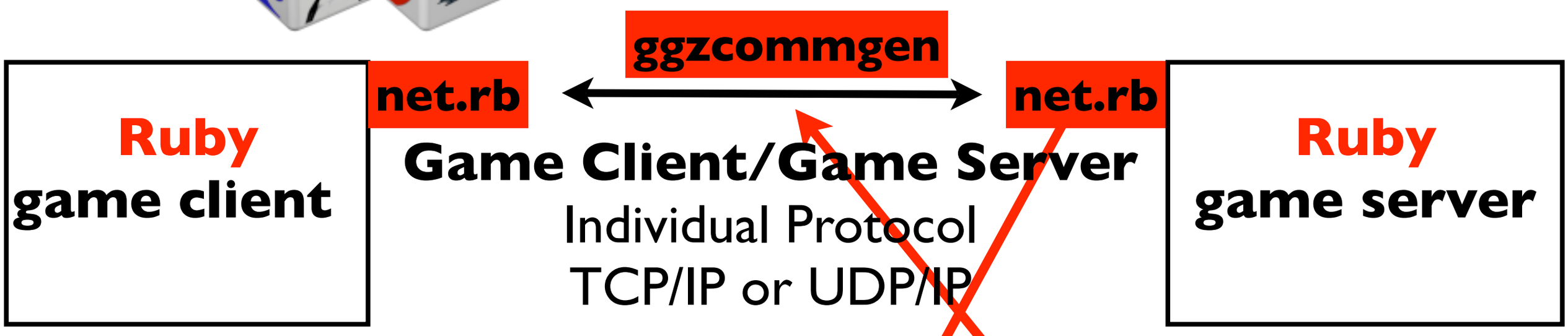
```
<ggzcomm engine="tictactoe" version="4">
  <definitions>
```

```
void ggzcomm_msggameover(GGZCommIO * io) {
   ret = ggz_write_int(io->fd, msggameover);
   if(ret < 0)
      ggzcomm_error();
   ret = ggz_write_char(io >fd,variables.winner);
   if(ret < 0)
      ggzcomm_error();
}
```

```
</ggzcomm>
```

Here's what the generated C code looks like.

GGZ Gaming Zone




```
<ggzcomm engine="tictactoe" version="4">
  <definitions>
```

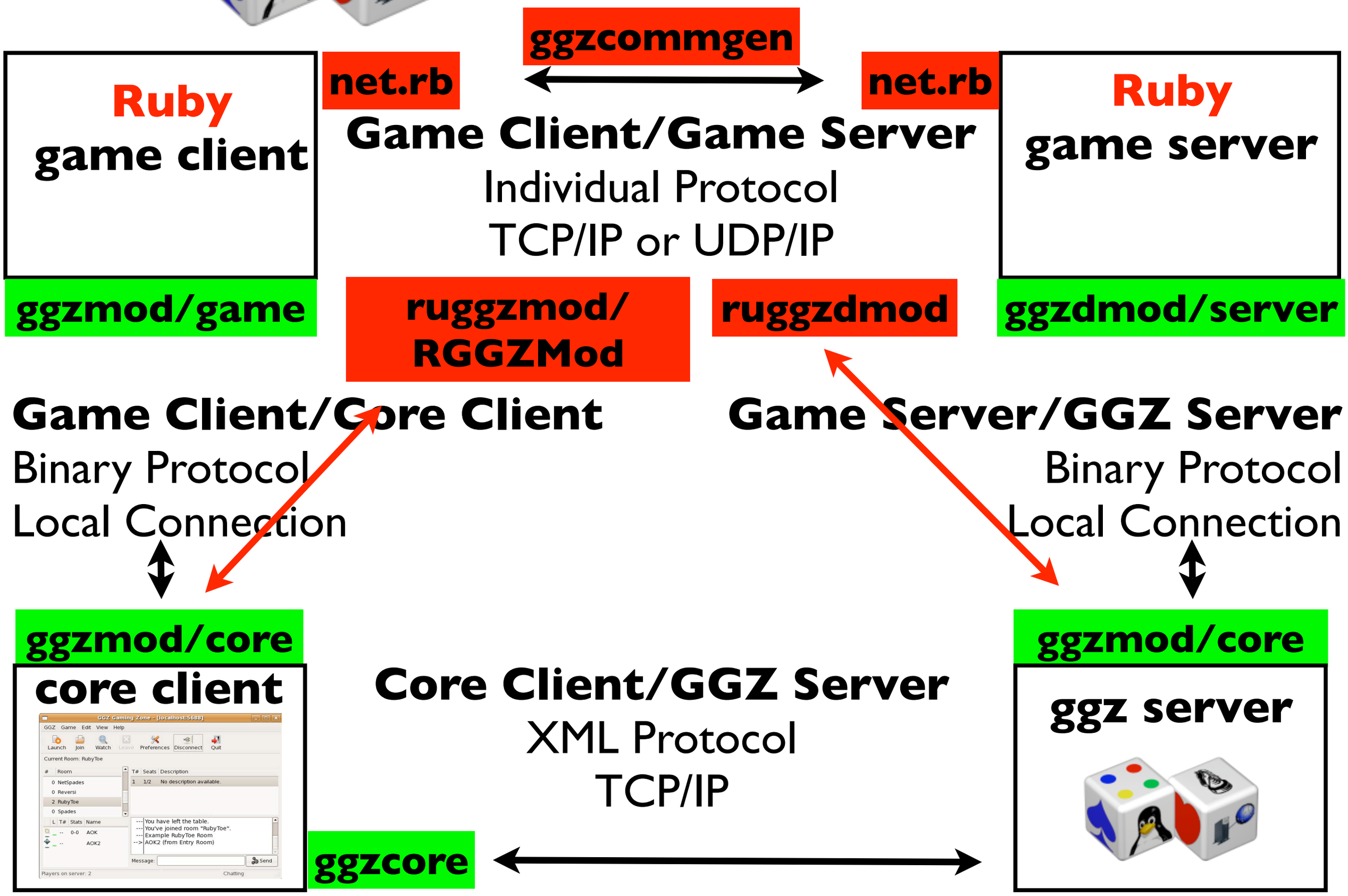
```
def send_gameover(p, winner)
  ...
  s = GGZRawSocket.for_fd($server.get_seat_fd(p))
  s.puti(::MSG_GAMEOVER)
  s.putb(winner)
end
```

```
  </server>
</ggzcomm>
```

So now that I've talked about what GGZ is, I can explain how GGZ has bolstered its support for Ruby.

Just last week, Josef Spillner, the GGZ lead developer enhanced the code generation utility so that it now generates proper Ruby code. Here's the generated Ruby code for the same send gameover method we just looked at in C. It gets a socket and sends the integer id code for the GAMEOVER message, as well as as the winner name in byte format.

GGZ Gaming Zone



Ruby bindings for the library that handles communication between the game server and the GGZ core server were recently added, as well as bindings for the library that handles communication between the game client and the core launching client.

There are two libraries in the red box representing communication between the game client and the core launching client -- one is the Ruby binding for the C library and one is a higher level pure Ruby library that wraps it -- to make it even easier to develop game clients with Ruby.

GGZ Gaming Zone

ggzcommgen

net.rb

QtRuby
game client

/Game Server

Protocol

UDP/IP

rubytoe
game server

ggz

/
GZMod

ruggzdmod

ggzdmod/server

Game Client/Core Client

Binary Protocol

Local Connection

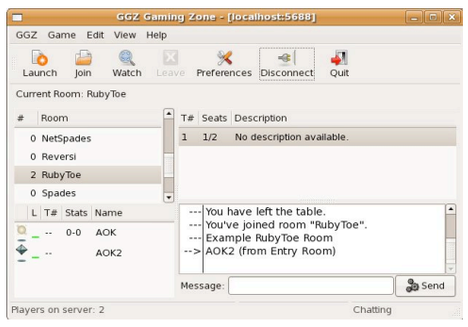
Game Server/GGZ Server

Binary Protocol

Local Connection

ggzmod/core

core client



ggzcore

Core Client/GGZ Server

XML Protocol

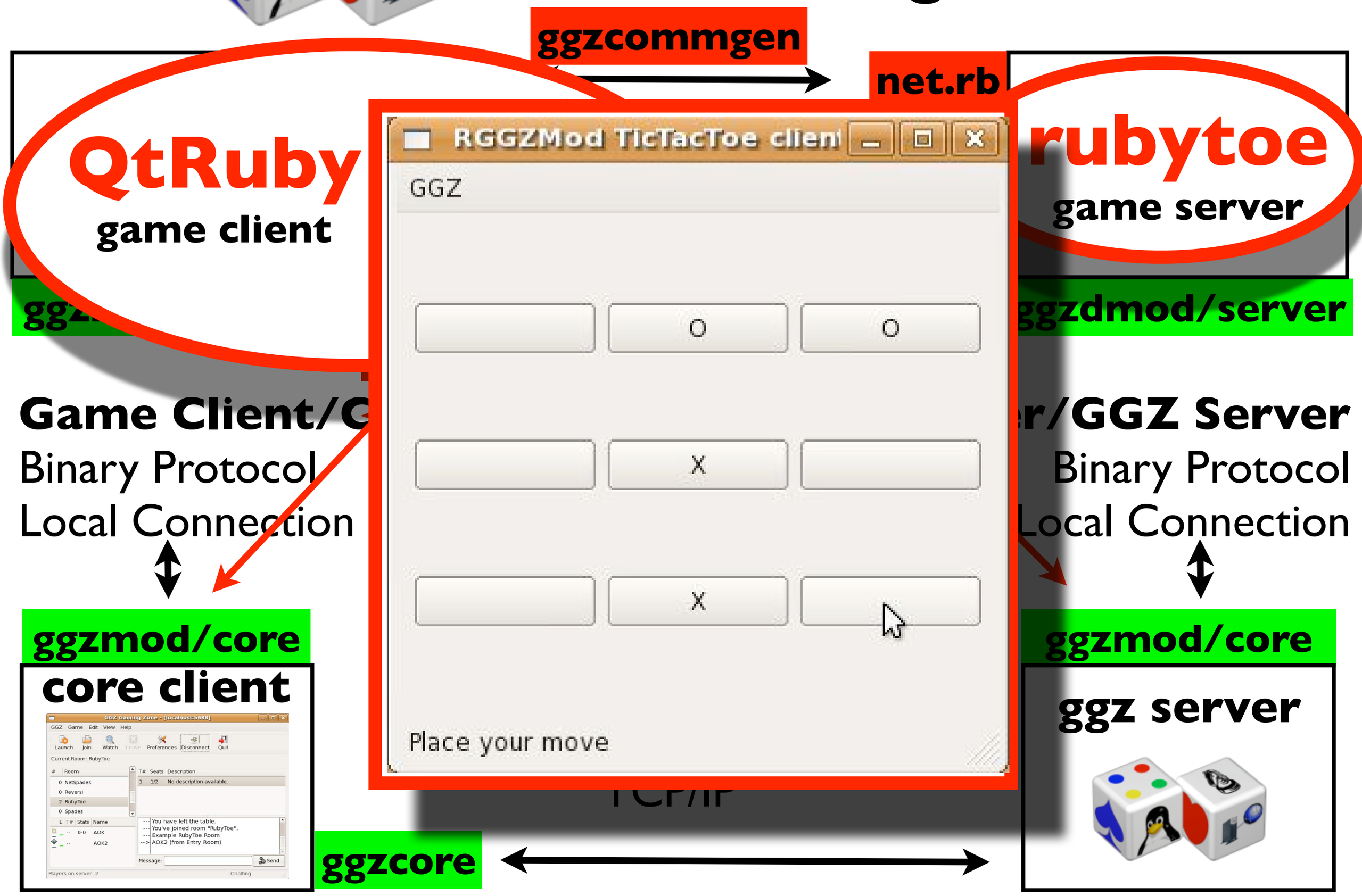
TCP/IP

ggzmod/core

ggz server

As a proof of concept, Josef wrote a tictactoe game server called rubytoe in Ruby, and a rudimentary client using QtRuby.

GGZ Gaming Zone



Here's a better view of the QtRuby game client.

GGZ Gaming Zone

rubytoe
game server

```
begin
   require "TTTAI"
   $ai = TTTAI.new
   puts "## Info: Using TTT-AI plugin ##"
rescue LoadError
   $ai = nil
   puts "## Warning: TTT-AI not found, using
internal random AI ##"
end
```

As a “proof of concept”, the RubyToe server does not implement all the features the C-based TicTacToe server implements, but one feature it does support is a pluggable artificial intelligence module. It uses the same AI module as the C version -- by virtue of another C extension.

This code fragment shows where RubyToe tries to load the AI module. If the AI module is not available, it just issues a warning that if a computer is one of the players -- it’s going to make a random choice based on the available squares.

GGZ Gaming Zone

```
begin
   require "TTTAI"
   $ai = TTTAI.new
   puts
intelli
rescue
   $ai =
   puts
interna
end
```

rubytoe
game server

```
def find_move(p)
   if $ai then
      return $ai.ai_findmove(p, 1, @board)
   end
   for i in 0..100
      move = rand(9)
      if evaluate_move(p, move) == ::MOVE_OK then
         return move
       end
     e
nd
    return -1
   end
end
```

This code block shows that if AI is available, its findmove function will be called when its the computer's turn. If not, the TicTacToe server will just choose a random number.

Both the experimental Ruby client and the experimental Ruby server are pretty basic -- but they signal the potential for a whole GGZ Ruby package. There's currently a GGZ python package, which is tightly integrated with Pygame. GGZ developers are thinking about integration with a Ruby game programming framework -- perhaps one of the frameworks we just talked about!

Is Ruby a legitimate player in the gaming space?

So is Ruby is a legitmate player in the gaming space?

Building Games with Ruby

Resources

GGZ Gaming Zone	www.ggzgamingzone.org/
Gosu	code.google.com/p/gosu/
Nebular Gauntlet	nebulargauntlet.org/
Ogre.rb	ogrerb.rubyforge.org/
Railgun	rubyforge.org/projects/railgun/
Rubygame	rubygame.sourceforge.net/
Ruby\SDL	kmc.gr.jp/~ohai/rubysdl.en.html
RUDL	sourceforge.net/projects/rudl/
Shattered Ruby	groups.google.com/group/shatteredruby wiki.shatteredruby.com

In a way I could have called this page “Is Ruby a Legitimate player inthe gaming space” instead of “Resources”.

Very definitely games that are fun to play can be created with Ruby.

One of the things that gives would-be game developers pause is Ruby’s garbage collection policy. Everything stops during garbage collection.

But that issue has certainly not stopped the developers of these frameworks in their tracks.

One way to deal with this might be to just write the server portion of your game in Ruby, maybe using GGZ’s libraries and protocols.

Shattered addresses the problem by calling gc every frame.

Julian Rashke, the lead developer for Gosu, recently tried modifying the library to call garbage collection every 10 seconds, but he took out that code before the latest release because he didn’t think it made a big difference. He has worked on improving Gosu performance, but garbage collection was not one of the issues he identified as major.

And it also did not stop a multimedia company from starting to use Gosu. The company is called Thinking Pictures. That’s my big news for the day. I can’t say anyone’s building a commercial video game with a Ruby game development frameowrk, but I can say a Ruby game development framework is being used commercially, which is a start.