Issue 25

September 2008

DEV.MAG

CREATE • DEVELOP • EXPERIENCE

CONTENTS

Find us on **Facebook!** Become a fan of Dev.Mag and help spread the word!

EDITOR
Claudio "Chippit" de Sa

DEPUTY EDITOR
James "Nighttimehornets" Etherington-Smith

DESIGNER
Quinton "Q-Man" Bronkhorst

CONTRIBUTORS
Rodain "Nandrew" Joubert
Simon "Tr00jg" de la Rouviere
Ricky "Insomniac" Abell
William "Cairnswm" Cairns
Danny "Dislekcia" Day
Andre "Fengol" Odendaal
Luke "Coolhand" Lamothe
Rishal "UntouchableOne" Hurbans
Gareth "Gazza_N" Wilcock
Sven "FuzzYspo0N" Bergstrom
Kyle "SkinkLizzard" van Duffelen
Chris "LionsInnards" Dudley
Herman Tulleken

WEBSITE ADMIN
Robbie "Squid" Fraser

WEBSITE
www.devmag.org.za

EMAIL
devmag@gmail.com

This magazine is a project of the South African Game.Dev community. Visit us at: www.devmag.org.za

Sarah was a pretty little girl, but she didn't live to see past the age of 7, much thanks to the HADRON COLLIDER MAKING THE ENTIRE UNIVERSE FOLD IN ON ITSELF! DEATH! SCREAMS OF TERROR! DEAAAAAAAAATH!

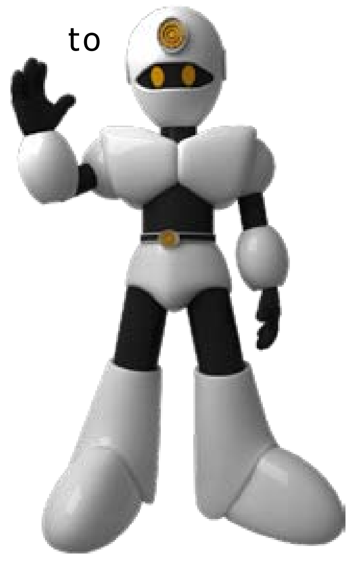
So things are getting a little frantic now. The local **rAge expo** is literally around the corner, and, in order to keep up traditional presence there we've had to shift up our release schedule by one week to have the next issue of the magazine ready by then. Which means we're all scrambling to make sure we'll have some great content to offer for that special issue; it also means that the magazine will be ready quite a bit earlier than usual. **Rejoice, dear reader, rejoice!**

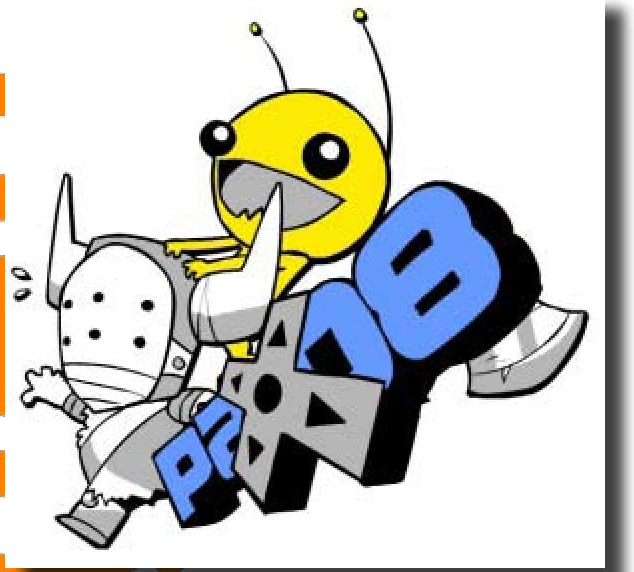
Additionally, there is a little under two weeks left before the **DreamBuildPlay** entry deadlines as I write this, and the teams are starting to feel the strain and pressure; fears about whether or not the games will actually be done in time niggle at the edges of our minds as final gameplay is tweaked and proper content is added. And all that means that I don't really have all that much to say this month that won't be tainted with the C# and sheep that are breeding in my mind at the moment.

Content for this issue is unfortunately a bit sparse as a result of the overly eloquent excuses listed above, but we haven't skimped on what we have. And that includes a positively gargantuan conclusion to the **Google App Engine tutorial** we started last month. And, because we're currently a little DBP crazy, we've got a nice informative tailpiece on how the competition turned out last year. But you'll certainly see all this more efficiently with a glance at the index page, so I'll leave it there.

That means it's about time I get back to making fluffy sheep... fluffy. Till next month, (which is actually the end of this one, but that's almost too confusing and daunting to think about right now) enjoy!

~ Claudio, Editor





Castle Crashers finally released on XBLA

http://www.castlecrashers.com/

After many delays, indie devs The Behemoth's newest title, Castle Crashers, finally joins Alien Hominid in their Xbox Live Arcade portfolio. Despite reports of persistent bugs that still plague the game even after its long trek through Microsoft's preliminary tests, the game is as fun as expected and its numerous references to internet culture show that it doesn't compromise on any of its roots. The game's soundtrack also primarily consists of content submitted to Newgrounds' Audio Portal.

DreamBuildPlay Warm-Up Challenge winners

http://www.dreambuildplay.com/main/winners.aspx

This year's DreamBuildPlay content was preceded by a 'warm-up challenge', focusing mostly on artificial intelligence and production quality, all to get developers into the swing of things for the main DBP contest. The winners of the Warm-Up Challenge were recently announced, netting themselves a small but not insignificant cash prize as well as potential opportunities at studies like Rare Ltd. and Lionhead Studios, who will interview the winning developers.

Knytt Stories bound for DS?

http://tigsource.com/articles/2008/08/28/knytt-stories-on-the-ds

A video recently floating around on video-sharing site, Youtube, has shown a dedicated fan's initial efforts in porting Knytt Stories to the Nintendo DS handheld platform. While it hasn't come very far, and it appears to be an unsupported venture and doesn't appear to have come very far yet, such gestures are a massive boon to developer morale.

IT'S HERE...

MULTIWINIA

SURVIVAL OF THE FLATTEST

Multiwinia: Survival of the Flattest is the latest addition to a stream of great games made by Introversion. If the name Introversion doesn't ring a bell, one might recall the likes of the hacking simulator Uplink, the retro revivalism of Darwinia and the ICBM fan club's favourite game, DEFCON.

Simon "Tr00jg" de la Rouviere

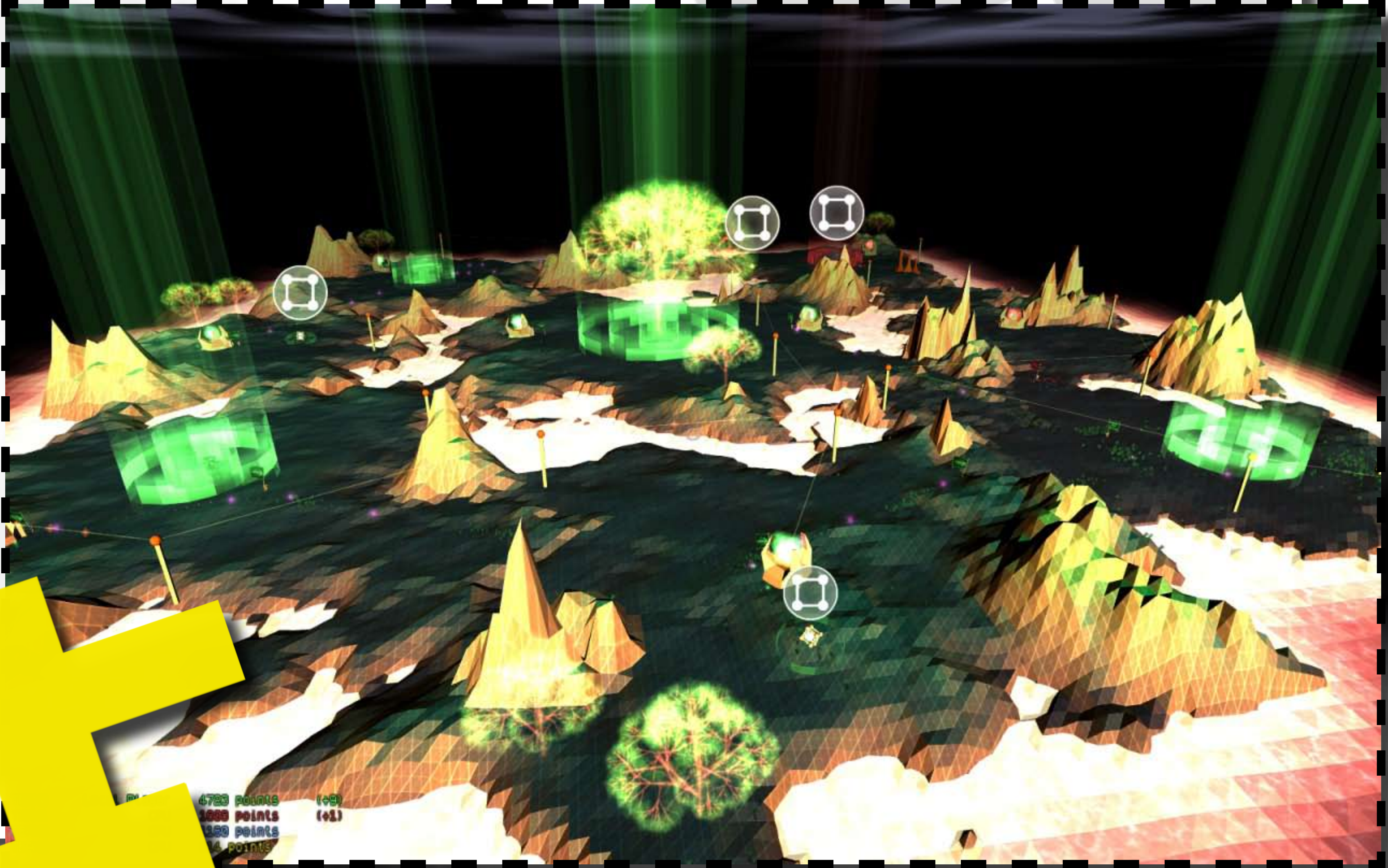
"HE KNEW BETTER THAN TO BE INVOLVED IN THE MESS OF WAR AND PROMPTLY ESCAPED!"

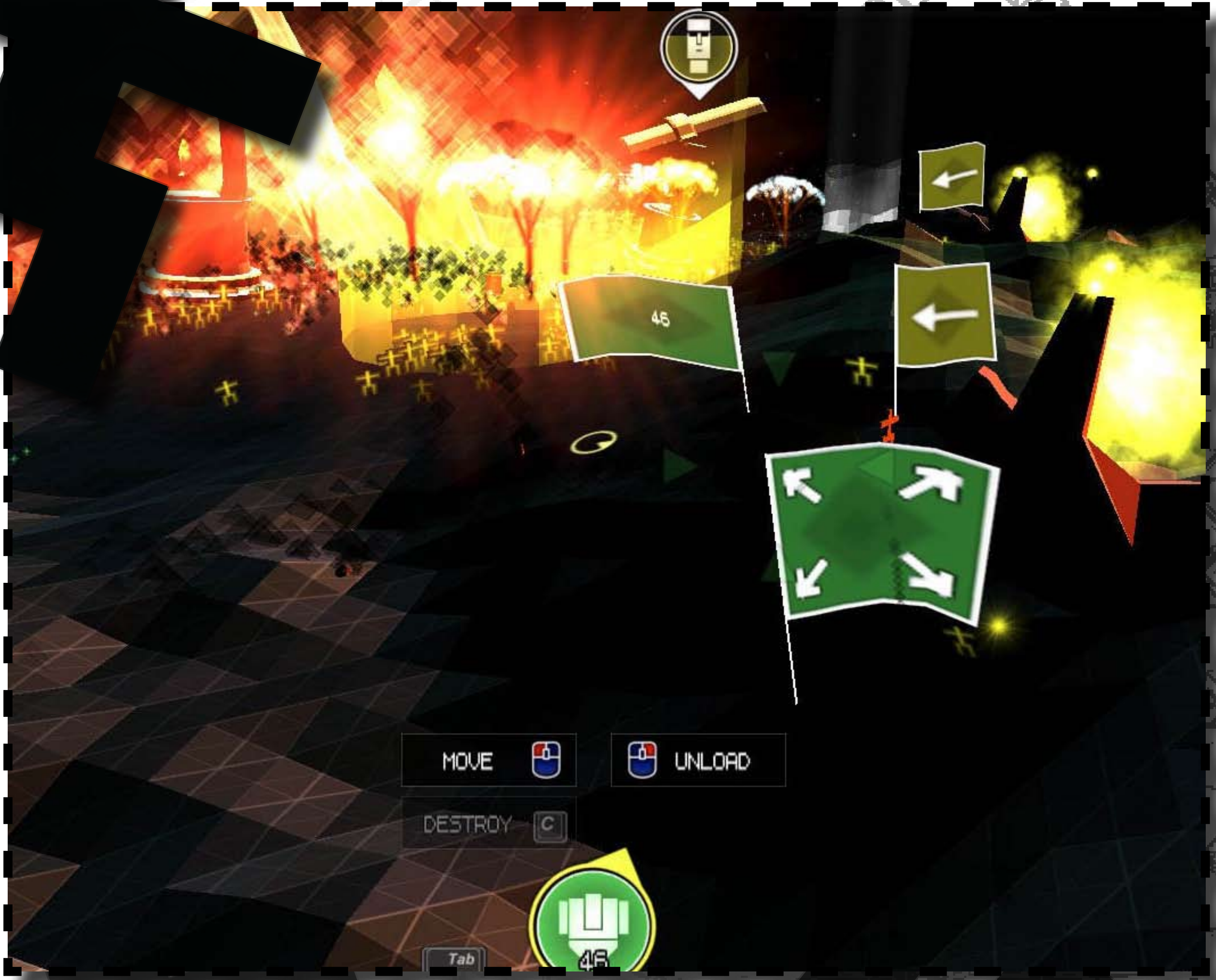
Sandwiched between an array of great games and a very promising lineup of future titles – which include the likes of Subversion and Chronometer – is Multiwinia, the multiplayer adaptation of Darwinia. After droves of Darwinia fans drooled over the concept of loads of bitty hominids destroying each other, their greatest wish was confirmed when Introversion announced Multiwinia early in 2007. Armed with the knowledge gathered during the creation of DEFCON and the acclaimed Darwinia, could Introversion possibly go wrong? Once the preview code landed on the Dev. Mag desk (well technically it arrived in the postbox), we were ready to find out! Apart from the preview code there came packaged a really neat blue spongy-type Darwinian. He knew better than to be involved in the mess of war and promptly escaped!

The game takes place in Darwinia, a virtual world created by Dr. Sepulveda. In this fractal world, live the peaceful and law-abiding virtual life forms called Darwinians. In Multiwinia's predecessor, the player fought against a virus that ravaged across the lands. After defeating the virus that infected their land of Darwinia, the Darwinians decided to do what the humans do: break up into factions and wage war on each other for (virtual) world supremacy!

All of the great elements of Darwinia are back, such the gorgeous fractal graphics. In the age of ultra-high-def graphics, it is a relief to see a game with some real style. The simple controls make a return, ensuring that commanding Darwinians is as easy as drawing one. With 'WASD' the camera is moved, and the rotation is done with the mouse. A ring expands when the left mouse button is held down; all of the Darwinians inside the ring will then be selected, ready to be issued commands.

There are no unit buildings, only spawn points. If the player is in control of one, Darwinians will start pouring out. A very useful feature is the ability to promote a Darwinian to give directions. If an idle Darwinian falls beneath the line of direction indicated by the promoted Darwinian, it will move that way. To sway the tide of battle, crates drop from the sky, carrying power-ups such as air strikes, reinforcements and turrets. It is strange to think of it this way, but that is all there is to Multiwinia; very simple and straight forward.





"THE DARWINIANS DECIDED TO DO WHAT THE HUMANS DO: BREAK UP INTO FACTIONS AND WAGE WAR ON EACH OTHER FOR (VIRTUAL) WORLD SUPREMACY!"

The preview code only had skirmishes in two of the game modes; King of the Hill and Capture the Statue. King of the Hill is a fairly traditional game mode; the longer a player holds a certain location on the map, the more points that player earns. In Capture the Statue, the player must fetch a statue on the map and take it back to their base.

The other game modes available in the full version are Domination; Rocket Riot; Assault; and Blitzkrieg. In Domination, one simply has to annihilate all of their opponents to win. In Rocket Riot, the player has to protect their rocket and capture solar panel arrays to power it up for launch. In Assault, one player starts behind a heavily fortified base. The players take turns to see who protects their base the longest. In Blitzkrieg the player must capture crucial flag points across the map.

Multiwinia doesn't start off slow like some other RTS games. It is a very frenetic affair right from the start. Considering the simplicity of the game, Multiwinia is very easy to pick up. Strategy fans looking for elements such as complex technology tiers and unit strengths and weaknesses won't find them with Multiwinia.

In most modern RTS titles, doing everything very fast usually ends in victory. Whether that constitutes a true strategy is another debate. With Multiwinia, it is refreshing to see a competitive RTS in which being tactical usually wins over being fast.

As mentioned, there isn't really much depth to the game and thus it is unlikely to see Multiwinia becoming something more than a casual multiplayer game. While Multiwinia is perfect for those coffee-break games, the game really comes into its own when played at a LAN with lots of players on one huge map. It is therapeutic to see armies upon armies of Darwinians duking it out across the gorgeous fractal world.

BLURRING The lines

LumaArcade is a prominent figure in South African game development, showcasing successful projects such as Mini #37. Dev. Mag managed to nab a few of the key developers; and in this, part 1, we chat to Luke Lamothe, focusing on their newest offer, BLUR, which will be available on InstantAction for free play soon.

Sven 'Fuzzyspoon' Bergstrom

Dev.Mag: As lead developer on BLUR, what facets of development did you enjoy most?

In terms of programming for BLUR, there wasn't a single aspect that really stood out as being enjoyable per se. As we were using a 'complete' game engine in the form of the Torque Game Engine (TGE), I wasn't really responsible for doing anything too exciting. Game development isn't all HDR and occlusion mapping! If I had to pick something, it would probably be a tossup between the fixes and tweaks to the physics and collision system required by TGE in order for as best a feeling, and behaving game as possible; and all of the little graphical additions that we made to TGE, like skid marks and ambient lighting lookup maps. As I wore a few hats during development, I would have to say that overall, what I enjoyed most was working with the great guys at GarageGames during the entire development process. It was really nice to work with industry professionals again, and especially ones who understand how a small studio works and can give us the freedom and responsibility to get things done as we feel best.

Dev.Mag: Conversely, what aspects had you pulling your hair out?

TGE's award winning networking. In all fairness, Torque has a great networking infrastructure that makes it uber-simplistic to get any game up and running in a networked environment almost from the word go. However, this networking system has some pretty severe flaws in it when it comes to dynamic physical interactions between high speed bodies, especially when their relative positioning is close; in other words, a car racing game. Luckily, someone at GarageGames had actually already developed improvements to the networking system that all but alleviate these issues, if used correctly, so I was able to integrate that code into BLUR, and after only a few more weeks of tweaking and fixing things, our networking was top notch!

Dev.Mag: In terms of integration with InstantAction, and the in-browser gaming; how was it different to 'normal' development?

Making a game for InstantAction doesn't require things to be that much different than creating a game normally. The most important thing is that the game is designed to be a multiplayer, arcade-like experience, which people can pick up and play and have fun with for 5 minutes at a time. Like a car racing game! In terms of design, the only real challenge is to make your game mouse friendly, as the game is run in an Internet web browser and all of your users will expect to be able to interact with your game with the mouse. In terms of art, you have to be very careful about the amount of resources that your game uses so that you can have the smallest download possible. Ideally, anything under 20 MB is a good target to aim for. I believe that BLUR is currently sitting at about 45 MB or so right now. It was over 100 MB before we converted all of our images to JPG and JNG. It is broken up into multiple packages that are downloaded in the background once a player first starts up the game and enters the menu system. In terms of programming, GarageGames provides an SDK which is used to interact with the InstantAction system, and for tracking the various states of the game, so that the game and the web browser are always talking the same language.

Dev.Mag: Which aspects of system layout and design changed due to InstantAction integration?

As I kind of touched on already, the biggest changes that we had to make were our menu system and the size of the game itself. BLUR's menu exists almost entirely now in JavaScript in your web browser. However, we really wanted to have the car selection screen in full 3D like it was part of the game (and not just some static webpage), so we worked with the guys at GarageGames to extend the functionality of their SDK. Our collective aim was to allow for game-based widgets to be active inside of InstantAction, which allowed us to keep our car selection menu as it was initially designed to look and function. In order for us to reduce the download size of BLUR, we first addressed the usual suspects; reducing texture dimensions as much as possible, cutting out textures that maybe weren't necessary, and reducing sound and music to lower fidelities. However, our biggest win came from using lossy compression on our textures, which I was initially against, as I was worried about the texture quality suffering. Luckily, after doing quite a few tests we found that all textures used on 3D models could easily use a 60% JPEG compression without any real noticeable quality loss in game. This was great for our BMP's that had no alpha channel, but for out TGA's with alpha, probably about half of our texture memory, we needed to come up with another solution. We tried using PNG's, but they are essentially just zipped TGA's, and as the game itself comes zipped, we didn't really get a savings by going to PNG. After talking to GarageGames, they pointed us towards JNG files which are basically PNG images saved with JPEG compression.

Once we implemented support for these into TGE and found a good compression scheme, we were able to save another very large amount of memory, upwards of 25MB I believe! Textures used for 2D GUI elements were another matter as any compression on them was fairly noticeable, so we just stuck with PNG compression or very lightly compressed JPG's for them.

Dev.Mag: For BLUR, how did the development cycle compare to your first racing title, Mini#37?

Due to the fact that we cut our teeth working on TGE by way of MINI#37, most of the development for BLUR was a much easier process. We did raise the bar a lot in terms of our art quality, and the physics behaviour of the game, so there was still a lot of hard work involved. However, as we now knew what to expect with TGE, we were much better prepared to handle any issues that came up, or to add support for features that we needed, in order to achieve what we wanted out of the game. Also, as BLUR was designed from day one to be a commercial release, it immediately had a much higher profile, and therefore expectations, than MINI#37 had. [MINI#37] was really only designed to be a branding experience in the form of a computer game that was free to play.

Dev.Mag: For aspiring developers, what tips could you pass on?

Always a tough question, no matter how many times I am asked it. All that I can really say is that if you have a passion for doing something, whether it is game development or something else, then you should stick with it and do what it takes in order to make a career out of it. Game development isn't a traditional career, especially in South Africa. The avenues that you can take to further yourself in this discipline are quite often few and far between unfortunately. Beginners, or people looking to get into game development, need to learn to start small and have patience with what they are doing. Nobody in the industry is where they are now because they only spent three months making a really rubbish space invaders clone. It takes years and years of study, self or institutional, and practice in order to learn what is necessary to succeed as a game developer. For people who have been doing this for a while now and who still love it, getting an actual job in the industry can be a difficult and frustrating process. My best advice for those people is to keep on making games and to keep on applying to game development companies if you have cool stuff to show them. Don't send them that rubbish space invaders clone that you made 4 years ago!

In terms of practice and portfolio, working in teams is a very important thing to try to do as it shows prospective employers that you can work in a team environment. It is quite different to being a one man show. However, the most important advice that I can give is that you need to finish your games. It doesn't matter if you are an artist, a programmer, a designer, or a sound guy. Employers in the game development industry want to see that you can stick to your guns and get the job done.

That's what he said...

The most exciting part of working on a game is always that initial phase where your ideas are coming together, you are doing new and interesting things, and you are seeing immediate results.

The most boring and difficult part is that last 10% when you need to fix all of your bugs, go over your game play with a fine-toothed comb, and tweak it until it is just right. Polish, polish, polish! I'll be the first to admit, that part of game development sucks. Like, seriously.

Unfortunately, a game isn't done until it is done, and without the experience of finishing off projects completely, you really won't be ready for what awaits you when you start to develop games for a living.

MONOCHROME

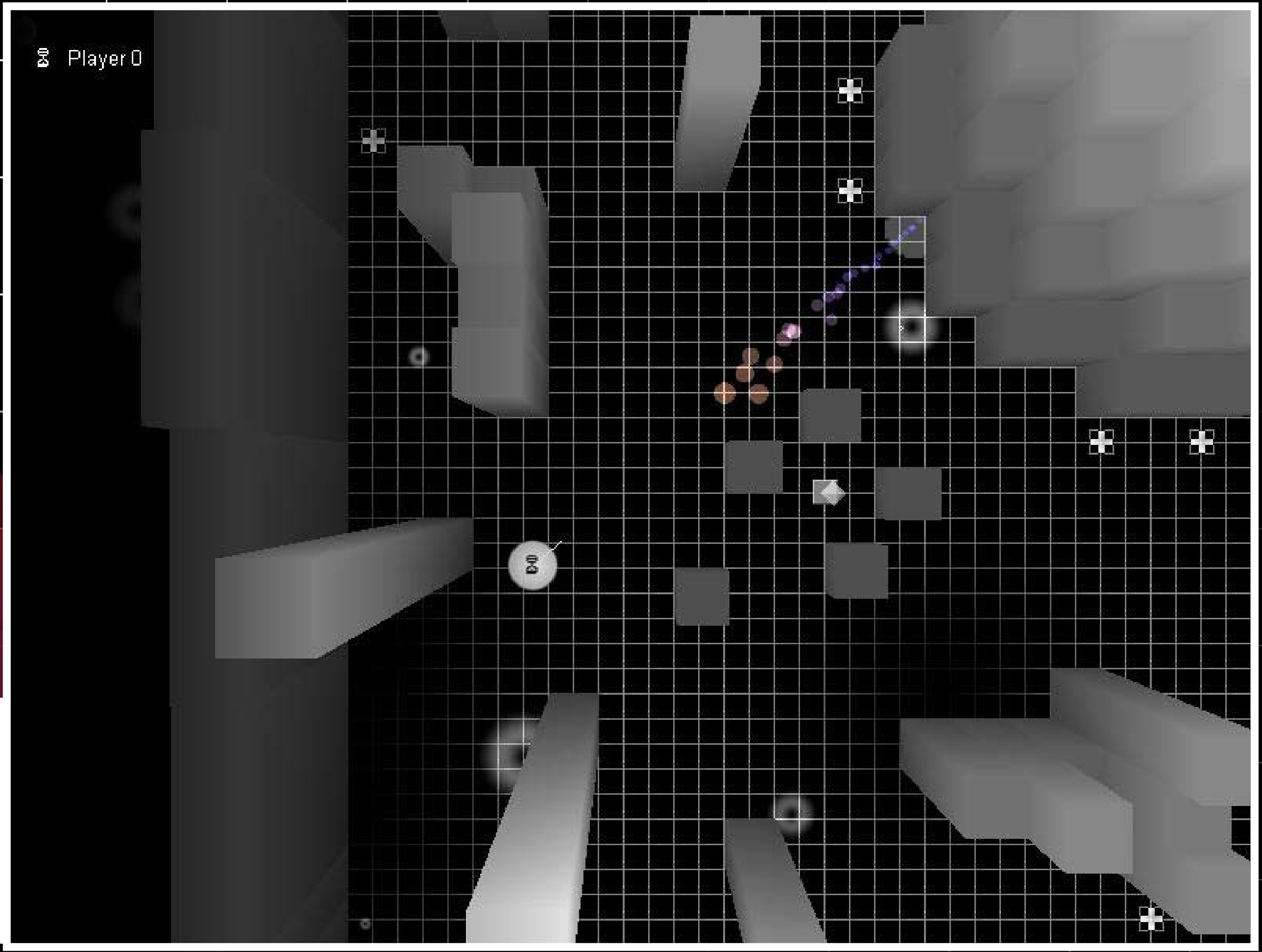
Have you ever felt the urge to kill your friends, but held back because those blood stains are such a pain to get out of the carpet? Well then, Danny "dislekcia" Day is once again to the rescue, with top-down multiplayer arcade shooter, Monochrome! You might also need to get some professional help, but Dev.Mag is not here to judge!

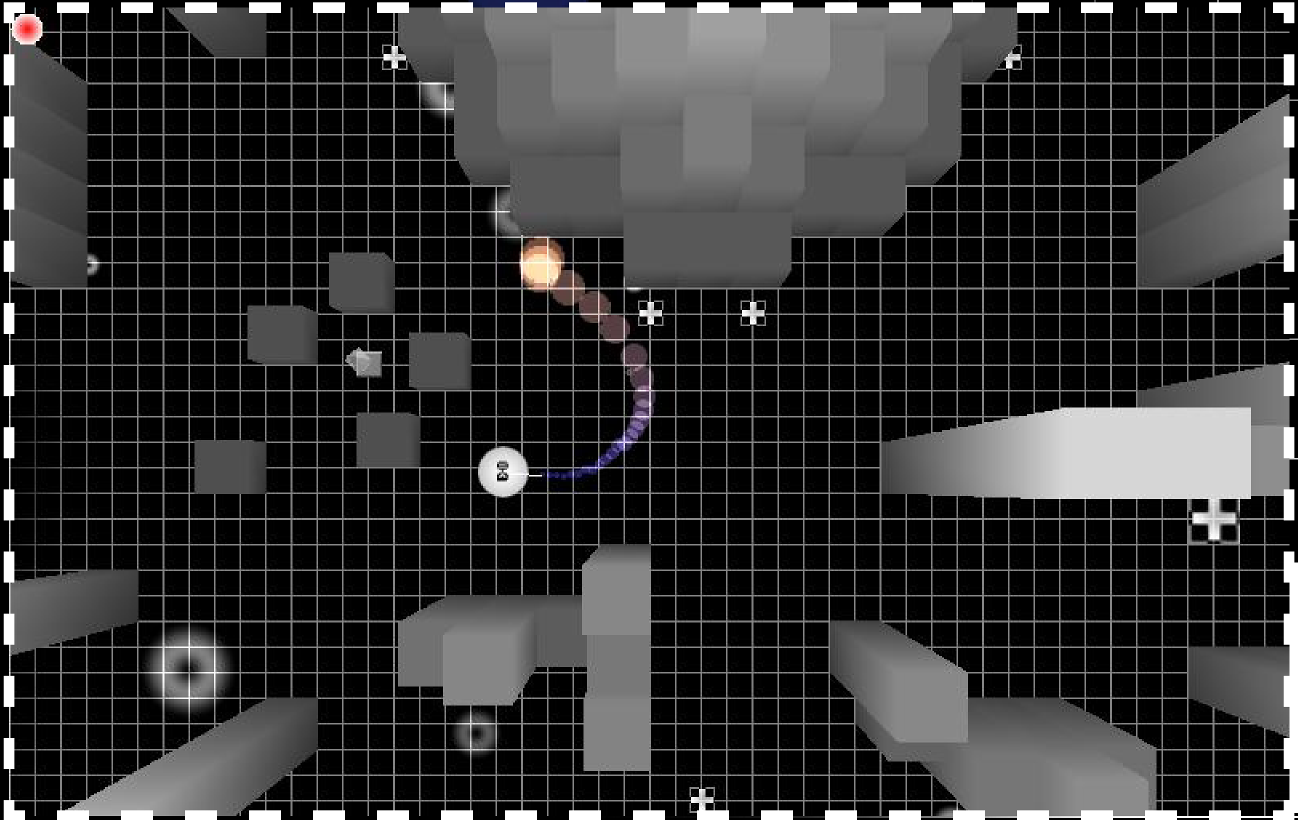
Chris "TheLionsInnards" Dudley

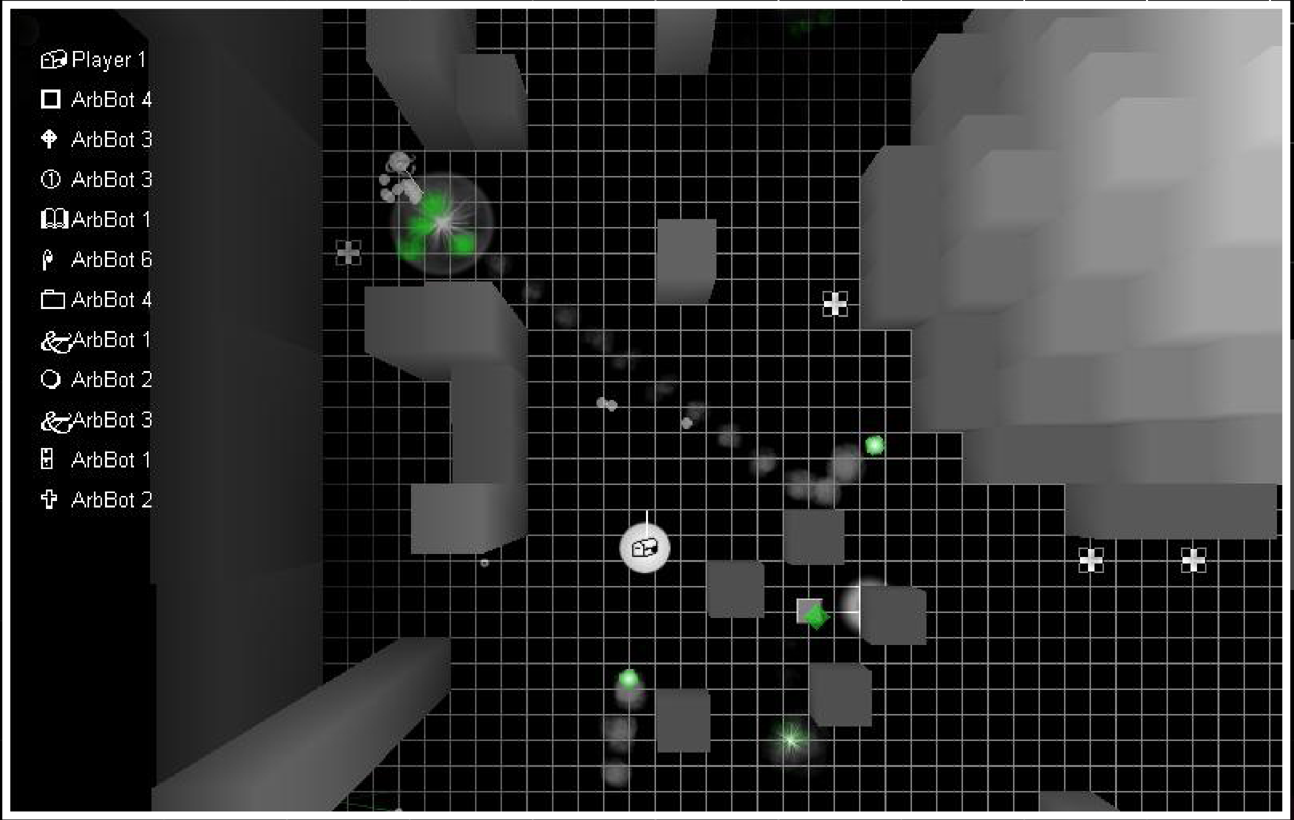
"IT BECOMES ESPECIALLY INTENSE AS DEADLY SNIPER ROUNDS AND ICE SHARDS WHIZ PAST."

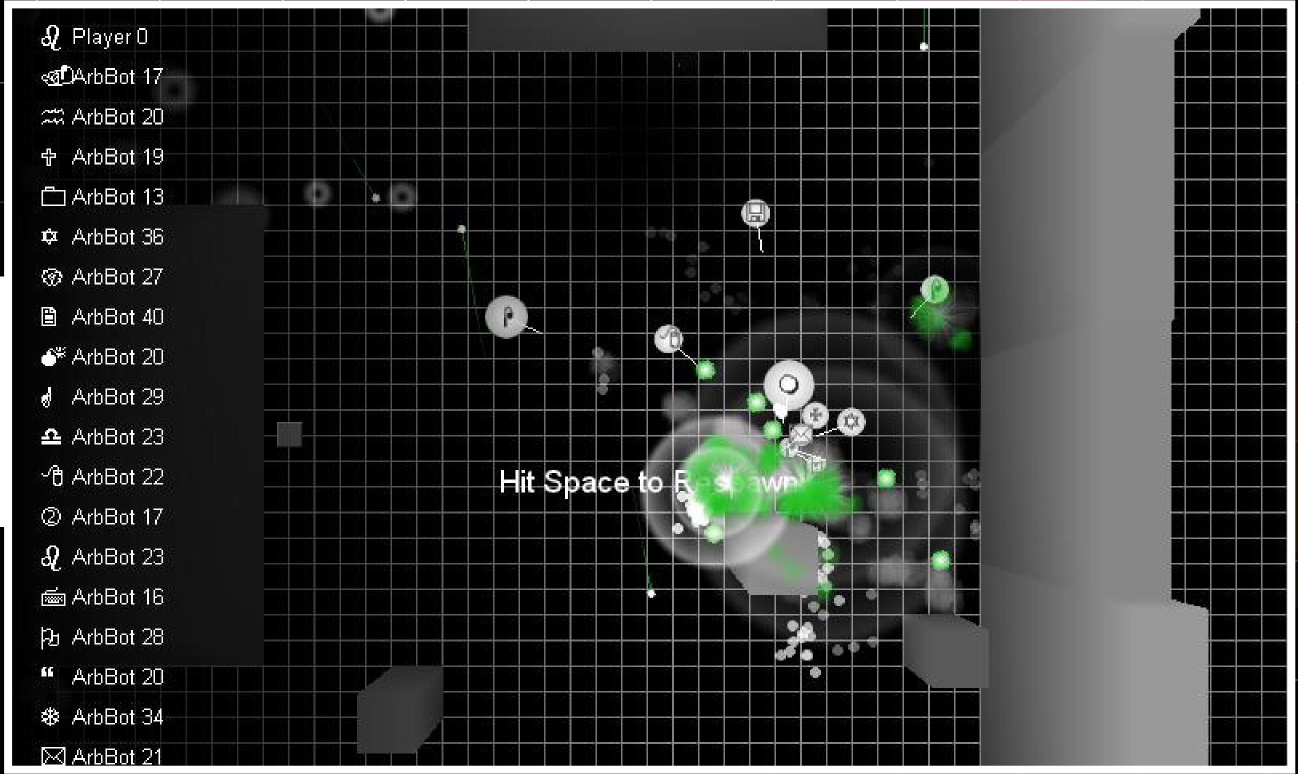
It is a difficult genre to stand out in, as there are a lot of reflex-heavy free shooters out there, but Monochrome manages to give a slightly more tactical experience by adjusting the gameplay according to the player's health. As the game starts, each player spawns as a disk and sets off to make good on all those pent up grudges held against their mates. When the carnage begins, and a player is hit, their avatar diminishes in size, allowing them to quickly slip away and seek a health pack.

Monochrome could be said to be 'frenetic,' in the same way as Michael Jackson might be described as 'a tad eccentric.' Projectiles fly everywhere and it becomes especially intense as deadly sniper rounds and ice shards whiz past. A couple of different weapons, including a flamethrower, a lightning gun and a gun that shoots bouncing bullets, mix up the gameplay nicely. The ability to leap out of harm's way, or in for a close quarters flamethrower kill means that one must be constantly aware of possible hiding spots and getaways.









The three available maps are quite bland, but one hardly notices the graphics. They offer a nice variety of open spaces for the ranged fire-fights and tight corridors for sneaky ambushes. Occasionally the walls of the maze-like maps obscure enemies, but it isn't a major issue, and can be used to ones advantage.

Playing by oneself will definitely get boring; the bots are fully functional, and are fine for a bit of practice, but the game's main attraction is to be able to relax with friends and de-stress while blowing each other away.

"RELAX WITH FRIENDS AND DE-STRESS WHILE BLOWING EACH OTHER AWAY."

IN CASE OF EMERGENCY
BREAK THE MOULD
www.pixofile.co.za
SOUTH AFRICAN ART COMMUNITY

MAKING GAMES WITH GOOGLE APP ENGINE PART 2

Herman Tulleken

In last month's issue, I explained how the Google App Engine works in broad strokes, and how you can use it for your games. Now it is time to get our hands dirty, and look at how an application is actually implemented. This tutorial explains how to make possibly the simplest game that can be made: the Guess the Number game. This tutorial covers a lot of ground. Typical of a web app, it uses lots of technologies, some or all of which you might not be familiar with. In a nutshell, this tutorial will show you how to program a Python application that generates HTML from Django templates and interacts with our datastore through GQL queries when the user requests specific URLs through his or her browser.

Before you Start

For this tutorial you will need:

- Python runtime (http://www.python.org/download/);
- A Google App Engine account (http://appengine.google.com/ – see last month's issue for some pointers before you sign up);
- The Google App Engine SDK (http://code.google.com/appengine/downloads.html);
- The files for this tutorial (http://code-spot.googlecode.com/files/code-spot_0.1.zip);
- Your favourite web browser and general purpose code editor;
- A working version of this tutorial is located at http://code-spot.appspot.com.

The zip contains these files:

- The application setup file (main.yaml);
- the Python server side code (main.py, game.py);
- some Django HTML templates (*.html), and;
- a style sheet (game.css).

Install Python and the GAE SDK. If you are unfamiliar with Python, learn it in 10 minutes from: http://www.poromenos.org/tutorials/python (provided you can already program in another language).
It is worth going through the SDK Getting Started Guide: http://code.google.com/appengine/docs/gettingstarted/.

Once you have everything installed, follow these steps:

1. Extract the zip file anywhere convenient (for example, C:\code-spot).
2. Rename the folder containing the extracted files to your GAE application name (for example, rename C:\code-spot to C:\my_app).
3. Open the main.yaml file with your code editor, and change the application name from code-spot to your GAE application name (in our example, my_app).

To test the application locally, from the command prompt, navigate to the application folder and start up the development server with the following command:

```
dev_appserver.py .
```

Note the space and dot at the end. This is the command for Windows – it should be similar on other operating systems. The dot simply denotes the current folder.

In your web browser, type in the following URL: http://localhost:8080/

Your web browser should now display the interface for the application.

To test your application on the actual server, from the command prompt, navigate to your application folder, and upload the application with the following command:

```
appconfig.py update .
```

Again, note the space and dot at the end. To test the application, navigate to the URL of your application, for example: http://my_app.appspot.com/.

Page Handlers

This project contains two handler applications. The one (main.py) is used to render the page for the root URL; the other (game.py) is used for the actual game. These files work entirely independently, which makes it easier to move the game, or add other applications to the same domain. So how does the application server know which python files to execute? It gets this information from the app.yaml file, which associates URL patterns with handler scripts, or marks them as static content. The handler application file is responsible for mapping specific URL's with Handler classes. The file main.py is a very simple example:

```
# main.py

#... imports...

class IndexPage(RequestHandler):
  def get(self):
    # This page does not require any variable values,
    # thus an empty dictionary will do
    template_values = {}

    # Calculate a new path
    path = join(dirname(__file__), 'index.html')

    # Render the template
    self.response.out.write(template.render(path, template_values))

###
# Handler mappings
###

handlers = [
  ('/', IndexPage)
]
#

def main():
  application = WSGIApplication(handlers, debug=True)
  CGIHandler().run(application)

if __name__ == "__main__":
  main()
```

The main function is the entry point of the application. It does two things: it constructs a new WSGIApplication instance, with the appropriate list of handlers; and it runs that instance. The list of handlers contains tuples. Each tuple consists out of a URL and a handler for that URL – a class that extends from RequestHandler. In this case, we have a single handler mapped to the root URL of our application. The handler defines a get function (this is executed when you go to the root URL in your browser, for example), that renders an HTML page from a template. Templates are discussed in more detail below. The HTML is written as the response, which means it would be served to the browser. The game application follows the same structure; the only difference is that the list of handlers is more elaborate:

```
handlers = [
  ('/game/', IndexPage),
  ('/game/clear', ClearPage),
  ('/game/hello', HelloWorldPage),
  ('/game/play', PlayPage),
  ('/game/test', TestPage),
  ('/game/finish', FinishPage),
  ('/game/your_best_scores', YourBestScoresPage),
  ('/game/overall_best_scores', OverallBestScoresPage),
  ('/game/new_game', NewGamePage)
]
```

Every URL that is part of the game has its own handler class, all of which extends (perhaps indirectly) from RequestHandler. Rendering a simple page (one that does not require any template variables), is normally so common, that the functionality has been put in a base class. Simple pages extend from this class (called SimplePage), and merely sets up the actual template to render. In addition to the Handlers, game.py also defines a data class (GameRecord), and some helper functions. Sections below describe these in more detail.

Django Templates

It is possible to write HTML directly. For example, the function that renders the root URL could be defined as follows:

```
class IndexPage(RequestHandler):
  def get(self):
    self.response.out.write('''
<html>
<head>
  <link type="text/css" rel="stylesheet" href="/stylesheets/main.css" />
</head>
<body>
  <p class="navigation">
    <ul class="navigation">
      <li><a href="/game/">Game</a></li>
      <li>1</li>
      <li>2</li>
      <li>3</li>
      <li>4</li>
    </ul>
  </p>
</body>
</html>'''
```

Although writing HTML code to the stream is convenient for simple pages, code can become extremely messy, very quickly, especially if you have pieces of HTML code that is reused frequently. A better alternative is to use an HTML template – a file that contains the basic HTML code that can be configured with variables (similar to embedded PHP). There are many template systems as it happens; GAE already supports Django templates – simple HTML files with Django markup in between. Below follows a crash course in Django templates. You can find a complete reference here: **http://www.djangoproject.com/documentation/templates/**.

Printing Variables

The value of a variable can be printed by putting it between double braces, like this:

```
<h1>{{ var }}</h1>
```

Note that the spaces surrounding the variable are compulsory.

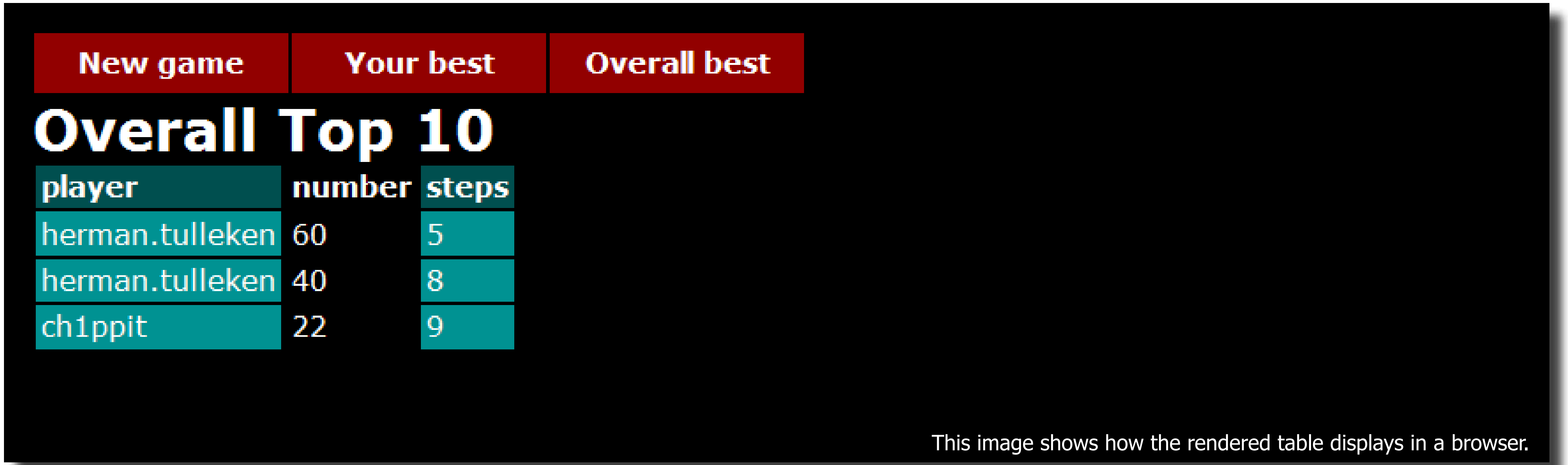
Simple Logic

The following snippet (taken from play.html) shows how conditional statements work in Django templates. This example will only render the error message if it is not empty:

```
{% if error_message %}
  <p class="error">
    {{ error_message }}
  </p>
{% endif %}
```

The example below (taken from records.html) shows how to use iteration to render a dynamic table. The variable records should contain a list; the syntax works the same as in Python:

```
<table>
  <tr class="heading">
    <td class="odd">player</td>
    <td>number</td>
    <td class="odd">steps</td>
  </tr>
  {% for record in records %}
    <tr>
      <td class="odd"> {{ record.player }}
      <td> {{ record.number }}
      <td class="odd"> {{ record.steps }}
    </tr>
  {% endfor %}
</table>
```



Overall Top 10

player	number	steps
herman.tulleken	60	5
herman.tulleken	40	8
ch1ppit	22	9

This image shows how the rendered table displays in a browser.

Inheritance

Every template can define named blocks. A template can inherit from another template, and override some of these named blocks. In the example below, the parent HTML sets up all the HTML that will be shared by all the files, and defines a block (called content) that can be overridden by children. The index file extends from base, and overrides the content block. Note that the index file does not contain the HTML and body tags – these are all defined in the root template:

```
<!-- base.html (the parent) -->
<html>
  <head>
    <link type="text/css" rel="stylesheet"
                 href="/stylesheets/main.css"
/>
  </head>
  <body>
    <ul class="navigation">
      <li><a href="new_game">New game</a></li>
      <li><a href="your_best_scores">Your
best</a></li>
      <li><a href="overall_best_
scores">Overall best</a></li>
    </ul>

    {% block content %}
    {% endblock content %}
  </body>
</html>
```

```
<!-- index.html (the child)-->
{% extends "base.html" %}
{% block content %}
<h1>Game</h1>
  <ul class="navigation">
    <li><a href="play">Play</a></li>
  </ul>
<p>
</p>
{% endblock content %}
```

Includes

Another way to reuse code is to include templates in other templates. The example below (overall_best_scores.html) includes the template, records.html, which is also included in, your_best_scores.html. Template variables defined in the outer template will also be defined in the included template.

```
{% extends "base.html" %}
{% block content %}
  <h1>Overall Top 10 </h1>
  {% include "records.html" %}
{% endblock content %}
```

How to render templates

Rendering templates is very simple; simply fill a dictionary with the variable that you expect in your templates, and call the render function with the template path and template dictionary. Here is an example that renders all the best scores in the game:

```
class OverallBestScoresPage(RequestHandler):
  def get(self):
    records = get_overall_best_scores()

    template_values = {
      'records': records
    }

    path = join(dirname(__file__), 'overall_best_scores.
html')
    self.response.out.write(render(path, template_values))
```

You almost always render a template in the get handler of a class.
Templates are not only for HTML

The nice thing about Django templates is that they can be used for any text format, not just HTML. That means, for example, that you can generate spreadsheets and SVG files directly from your data.

Defining Data Classes

Making a data class that will work with the record store is easy. Here is the data model for a game:

```
class GameRecord(Model):
  player = UserProperty()
  active = BooleanProperty(default=True)

  number = IntegerProperty()
  steps = IntegerProperty(default=0)
  guess = IntegerProperty()
```

Any class that will be put in the record store must do two things:

1. extend from Model (a class in google.appengine.ext.db), and;
2. define class variables as properties that will be the fields stored in the data store for this kind of record.

The second item needs some explanation. Class variables are similar to static variables in other languages – that is, they are associated with a class rather than an instance of that class. In Python though, it is possible, through a bit of magic, to make class variables change the way instance variables behave. In this case, the Google API authors set things up so that a class variable defines the type of an instance variable, and moreover write and read the instance value from the data store when required. This means, you do not have to worry about how these records are managed behind the scenes. Once you have defined your class, you can simply use it as any other Python class, as long as you remember to call the "put" or "get" commands as required. Here is the function that makes a new GameRecord, and commits it to the database:

```
def make_new_game():
  active_game = GameRecord()

  active_game.player = get_current_user()
  active_game.number = randint(1, 100)

  active_game.put() #commit to database

  return active_game
```

Queries

Google provides a SQL language with which queries can be made (called Google Query Language; GQL):

```
def get_your_best_scores():
  """Returns the ten best scores of the current
player."""
  query = GqlQuery('SELECT * FROM GameRecord '
           'WHERE player = :1 '
           'and active = :2 '
           'and steps >= 0 '
           'ORDER BY steps ASC',
           get_current_user(),
           False)

  records = query.fetch(10)

  return records
```

Note that every query needs to be fetched after it is constructed. The fetch method takes a number, which is the maximum number of records the fetch will return. No fetch can return more than a 1000 queries. Explaining the full syntax of the query language falls beyond the scope of this tutorial. This example selects all the records where the player is the current user, the game is active (i.e. not done), and the number of steps taken so far is positive. For more detail on GQL consult this reference: **http://code.google.com/appengine/docs/datastore/gqlreference.html.**

Getting data from post requests

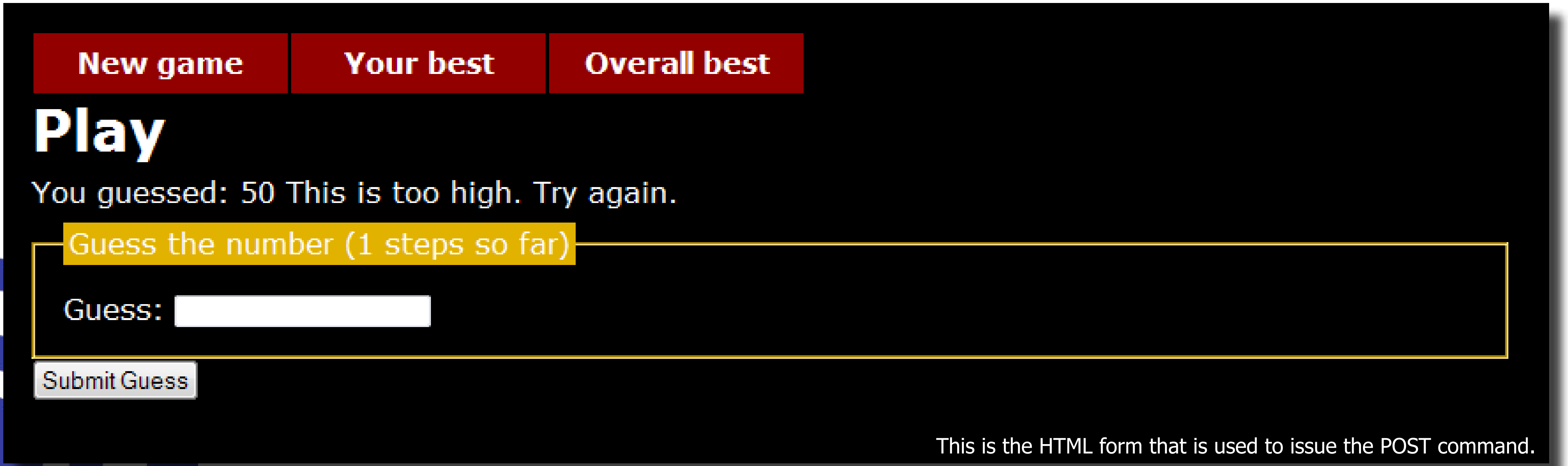
The player input for the game is obtained with an HTML form, whose data is posted to the PlayPageHandler. In the post method of this handler, the following piece of code retrieves the data:

```
def post(self):
  active_game = get_active_game()

  try:
    active_game.guess = int(self.request.
get('guess'))
    #...perform some game logic...

  except ValueError:
    self.redirect('/game/play?error=' +
      str(ERROR_NOT_A_NUMBER))
```

Note that the try-catch block surrounds the code, in case the user enters a non-numeric string. If this happens, we re-request the page (with GET), but with an error parameter set (see below). POST commands are usually issued from HTML forms, with their action set to POST. The command is issued when the user presses the submit button.


New game
Your best
Overall best

Play

You guessed: 50 This is too high. Try again.

Guess the number (1 steps so far)

Guess:

Submit Guess

This is the HTML form that is used to issue the POST command.

Getting data from get requests

The following code snippet shows how to retrieve data from a get request (PlayPageHadler's get method, in game.py):

```
error = self.request.get('error')

if error:
  error_message = error_messages[int(error)]
else:
  error_message = None
```

For example, if we get the following URL: /game/play?error=0 the piece of code above will assign the string '0' to error, and then the string 'You must enter a number!' to error_message. When we simply do not assign an error number (/game/play), error is assigned None, and so is error_message.

Remember to set up static files:

It is easy to forget this. Whenever an image, or other media type file does not display, check that it is in a static content directory; as set up in the app.yaml file.

Check that you always put data after you modified it:

Another easy mistake to make is to modify a record (by reassigning properties), but then forget to call the put method on that record. Get into the habit of always checking that this method is called after data has been modified.

Use the following URL to access the local admin console:

When the server is running locally, you can access the admin panel with the following URL: http://localhost:8080/_ah/admin

It is not as powerful as the online version, but is still very useful.

Define your own Property classes:

You can define your own Property classes that extend from the Property classes supplied. This is useful when you want to:

- treat certain properties differently from others (in formatting them on screen, for example), even though they have the same type, or;
- to add some extra description data with properties.

Use function decorators for checks before processing commands:

Function decorators are great for doing a bit of pre-processing before processing HTTP commands. For example, the following function decorator will check whether the user is logged in, and if not, redirect the user to an error page. Otherwise, the get is processed as usual. The example also shows how the decorator is applied to the get method of the YourBestScoresPage. Note that the decoration is done where the function is defined (not where it is called); this means that once you have decorated the function, you can always be sure the decorator will be executed, no matter where the function is called:

```
def check_logged_in(fn):
  """This decorators redirects the user to
an error page if
He / she is not logged in."""
  def wrapper(self):
    if get_current_user():
      fn(self)
    else:
      self.redirect(create_login_url(self.
request.uri))
  return wrapper

class YourBestScoresPage(RequestHandler):
  @check_logged_in
  def get(self):
    records = get_your_best_scores()

    template_values = {
      'records': records
    }

    path = join(dirname(__file__), 'your_
best_scores.html')
    self.response.out.write(render(path,
template_values))
```

Define a debug page which allows easy access to debug functions:

Debugging a web app can be made a lot easier if you define a page that contains links that will:

- give you special views of your data;
- fill your database in bulk, and;
- clear your entire database.

Make sure that you (and other developers on the project) are the only ones that can access this file. Take special precautions to make sure that these functions are not called in the production version – just imagine debugging the live version and accidentally deleting all your users' information!

Be careful not to corrupt your database:

Once you start collecting real data, it is important to prevent making changes that will affect the structure of the datastore. It is very easy to corrupt your database, making it necessary to either write code to correct the error, or to delete entries from the datastore. As an example: when working with a string property with choices (as can be used in a drop-down list), changing the choices can corrupt the datastore – when an existing element has a property defined as a choice that does not exist after the change. If you do not immediately spot this mistake, entries can be added that uses the new type, making it hard to revert!

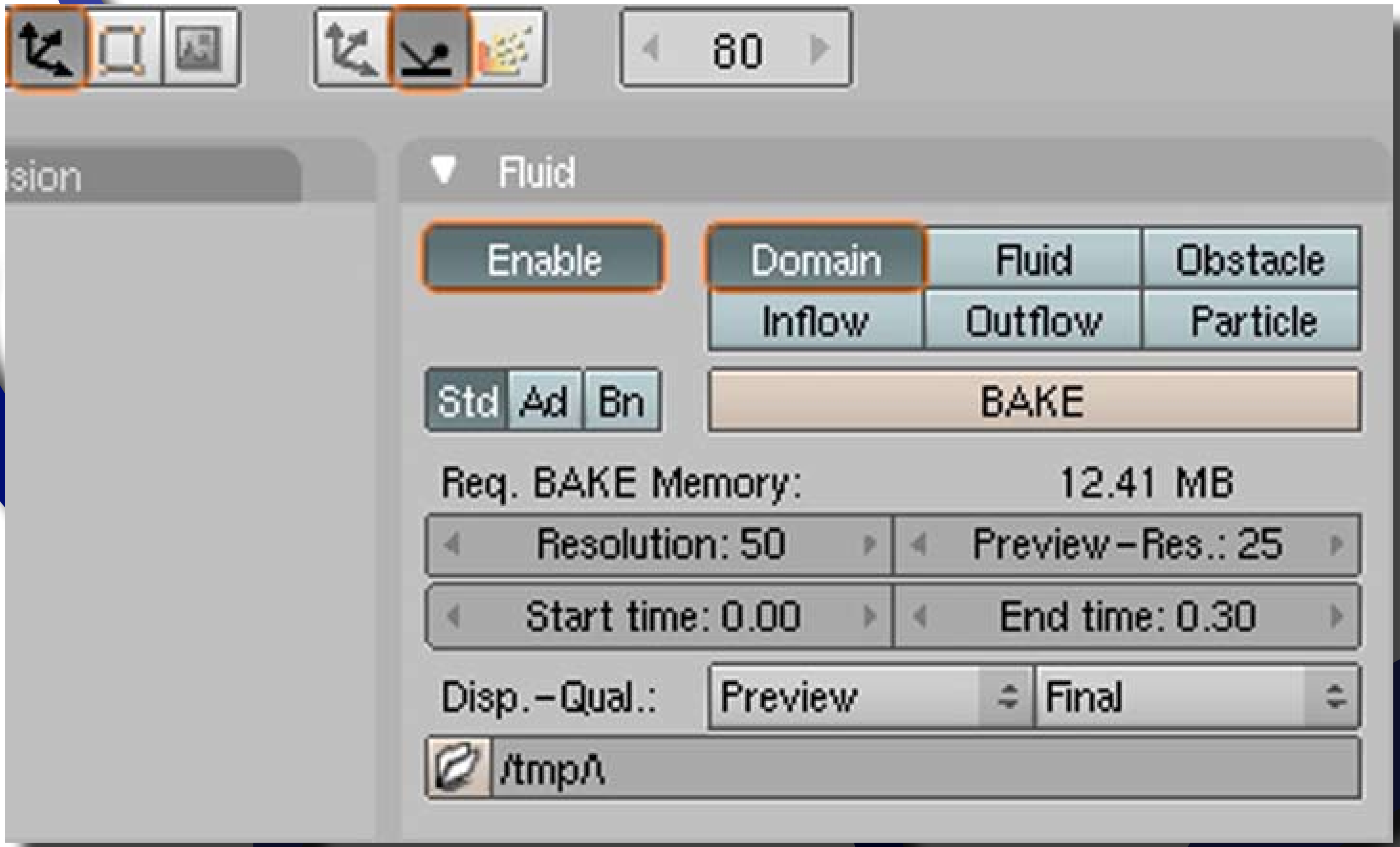
Object Pascal has a lot to offer...

www.PascalGameDevelopment.com

Blender's fluid simulator is quite possibly the most complex and powerful feature in its simulation arsenal. It is capable of modelling the behaviour of liquids of variable viscosity; so anything from the flow of water to that of lucent toxic sludge can be simulated fairly accurately, and often in quite a visually spectacular fashion.

BLENDER FLUID DYNAMICS

Claudio "Chippit" de Sa



80

Req. BAKE Memory: 12.41 MB

Resolution: 50 | Preview-Res.: 25

Start time: 0.00 | End time: 0.30

Disp.-Qual.: Preview | Final

/tmp/

Initial setup

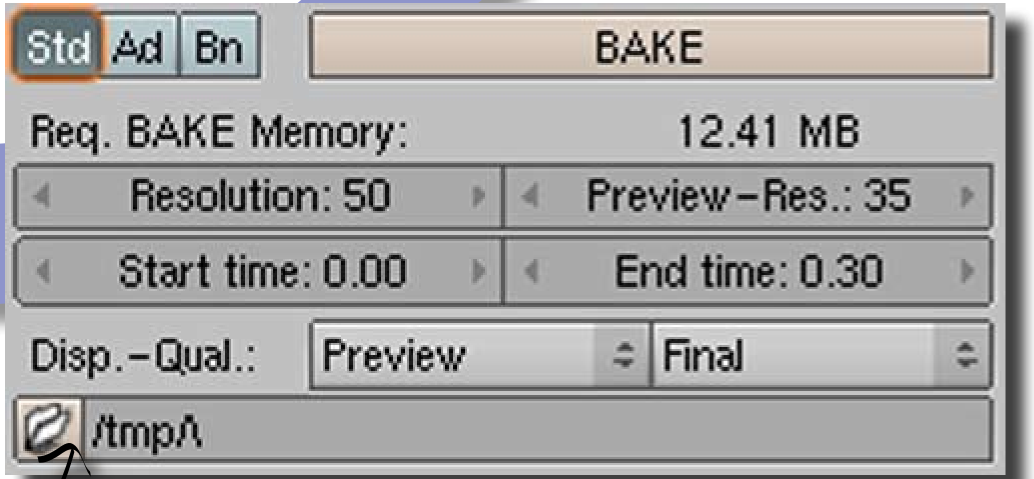
The setup of the fluid system is a different and slightly more cumbersome process in comparison to the other simulations. The most important part of this procedure is defining the fluid domain. Imagine this as a fixed, invisible box in which all fluids in the simulation will be contained. No fluids can pass outside the box, and all objects that will interact with the simulation must be placed within its bounds. To define the fluid domain, create a cube object and then, in the Physics Buttons' fluid tab, enable the Fluid button, then the Domain button. This tells Blender that the cube will be involved in the simulation by serving as the encompassing domain.

It's important to note that the domain object will actually become the fluid itself when the simulation is done. Therefore the material you wish the fluid to be drawn with must be applied to the domain object itself. At the moment, every blender project is also limited to a single domain object.

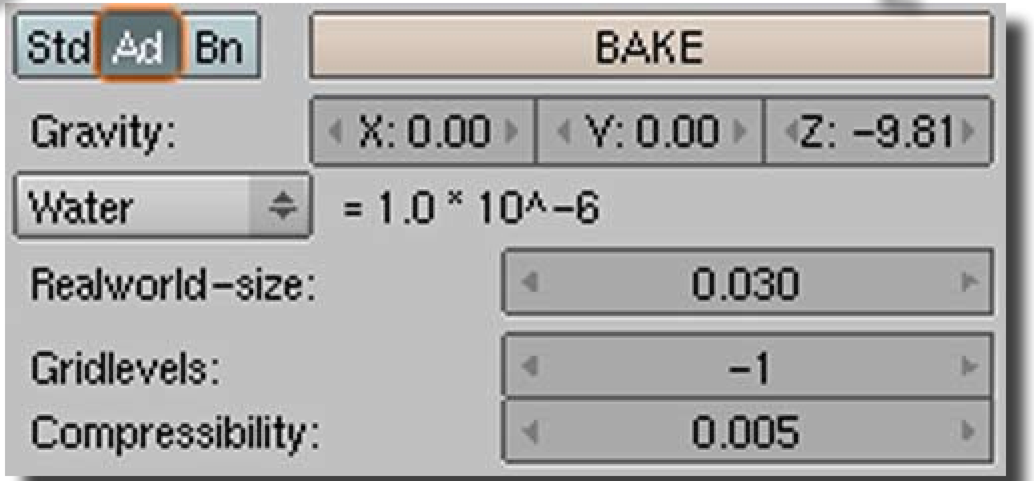
Domain settings

The domain object contains the most important settings for the simulation. These are divided into 3 subcategories, toggled with the Std (standard), Ad (Advanced), and Bn (Boundary) buttons on the fluid tab. Important settings will be explained below.

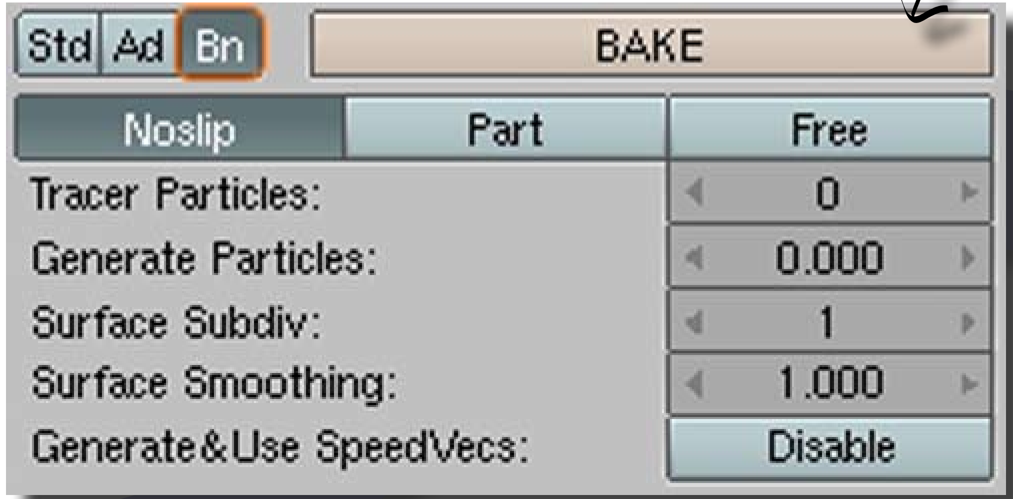
The most important option in the simulator is the Resolution setting. The fluid domain is essentially three-dimensional grid, and the resolution setting defines the size of the grid along the domain's longest axis (or all of them, if the domain is cubic). A higher setting means smaller grid units resulting in a more accurate simulation. The amount of memory required for the simulation increases exponentially with this setting. Make sure it is never set such that the required memory higher than your available system resources; this may cause unexpected behaviour such as extreme slowdowns or crashes. The Preview-Res setting defines the resolution of the preview mesh that can be displayed in the client or used for test renders, but doesn't affect render times or memory requirements. It must be smaller or equal to the Resolution setting.


Std Ad Bn BAKE
Req. BAKE Memory: 12.41 MB
Resolution: 50 | Preview-Res.: 35
Start time: 0.00 | End time: 0.30
Disp.-Qual.: Preview | Final
/tmp/

The Start time and End time settings define the time span of the simulation in seconds. The difference between the two values is the amount of 'fluid-time' that will be simulated over the duration of the animation, as set by the starting and ending frames in the Anim tab, under Scene buttons. The two Display-quality settings define whether the full resolution mesh is used in the graphical user interface and in renders.


Std Ad Bn BAKE
Gravity: X: 0.00 Y: 0.00 Z: -9.81
Water = 1.0 * 10^-6
Realworld-size: 0.030
Gridlevels: -1
Compressibility: 0.005

The advanced tab exposes a few less frequently used options, such as the ability to manipulate gravitational forces on the liquid in all 3 axes, as well as setting the viscosity of fluids in the domain. The most important option here, however, is the Realworld-size setting. This controls the size of the largest domain axis in meters, defining how the liquids inside the domain should behave.


Std Ad Bn BAKE
Noslip | Part | Free
Tracer Particles: 0
Generate Particles: 0.000
Surface Subdiv: 1
Surface Smoothing: 1.000
Generate&Use SpeedVecs: Disable

The boundary tab lets you define some settings relating to the domain itself. The Noslip, Part, and Free settings control how much friction will be experienced by liquids colliding with the domain boundary. The other notable option is the Generate particles field, which will create additional particles for splashes, greatly increasing the visual effect. For this to work, however, the Surface subdiv setting must be 2 or greater. This is similar to the standard subdivision modifier, but is processed slightly differently for better results on the fluids and, as such, is preferable.

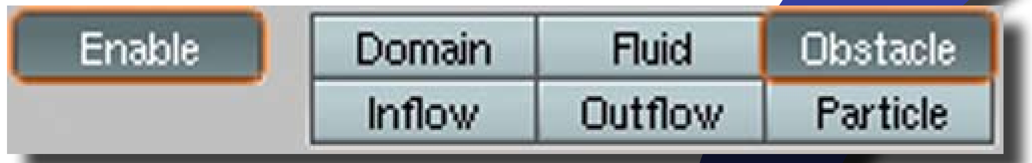
Actual simulation

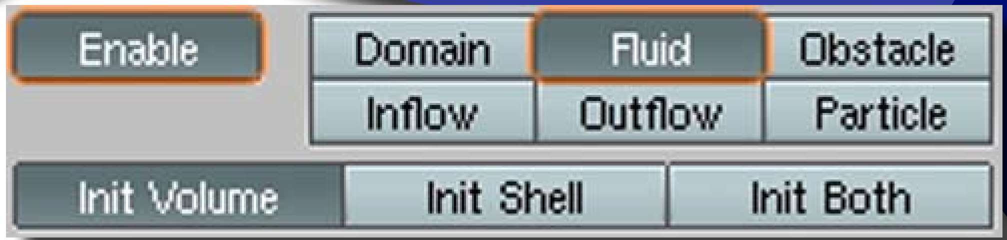
Now for the fun bit. Create any solid mesh object (it should have volume, so planes and circles won't do here), and place it inside the domain cube. This mesh will become the fluid in our domain. Scale it down a little bit so that it doesn't fill a significant portion of the domain volume, and, when you're satisfied with it, enable fluid simulation, and click the Fluid button. You can use Inflow for this purpose too; the difference between the two is that inflow will constantly generate more liquids for the duration of the animation. This can potentially fill up a domain quite quickly.

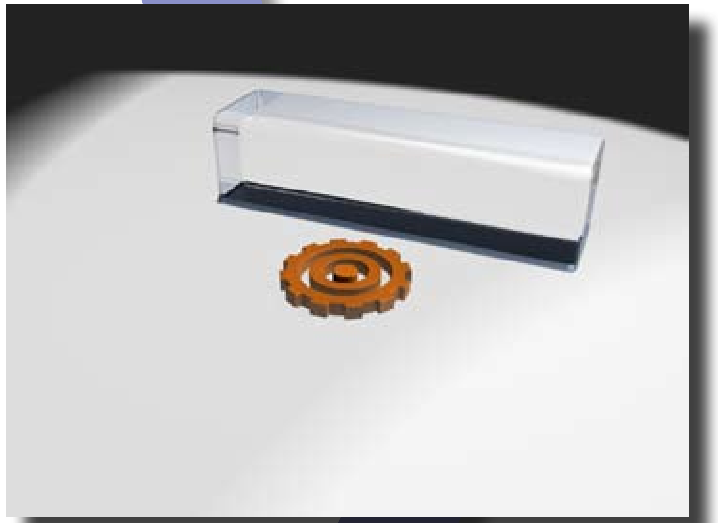
Once you've added and set up your fluid object, select the domain, and click bake. Be warned that this is a lengthy process, though it can be cancelled with the escape key at any time. When you're done, use ALT-A, or the arrow keys to step through the animation and see the results. When you render it, you'll want to move the mesh you've just created to a different render layer so that it isn't drawn and doesn't interfere with your output.

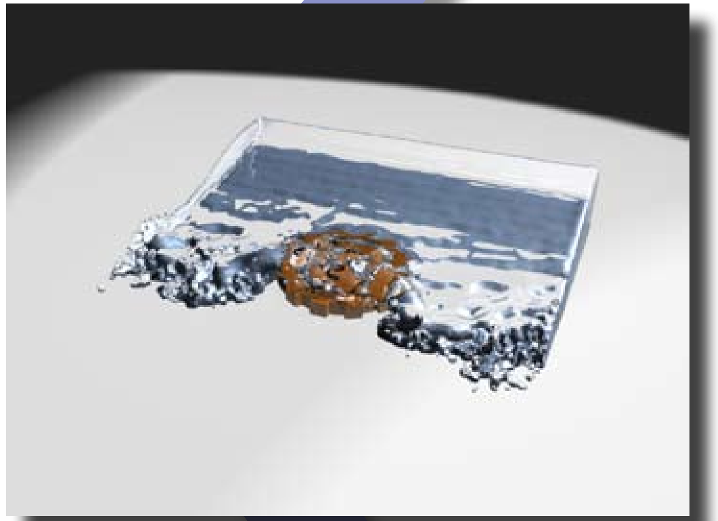
In order to make things slightly more interesting, we're going to add an object for the fluids to collide with. Add another solid mesh object, move it into domain space, then enable Obstacle in the fluid settings tab. You'll notice that obstacles share the Noslip, Part and Free friction settings of the domain boundary. Obstacles can also use keyframe animation settings for fluids. Once you're satisfied with the obstacle, select your domain and bake the simulation again. Check your output and tweak as necessary.

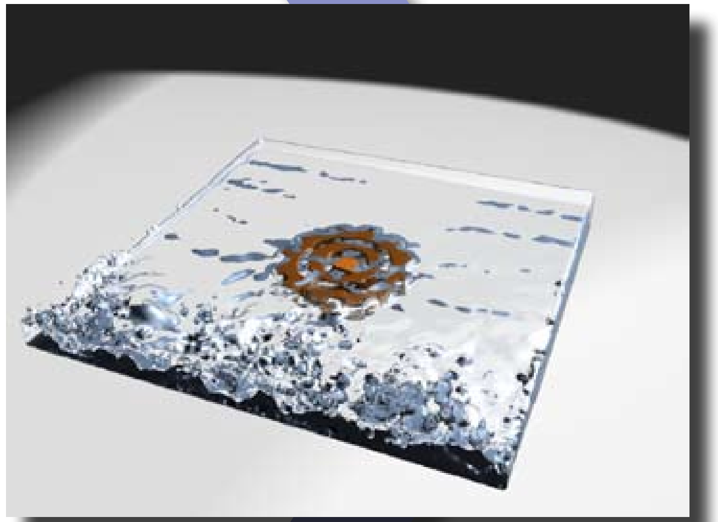
Once you're satisfied with the scene, it's time to ramp up some settings for quality. Depending on the size of your domain and the required quality, the resolution setting needs to be increased to anything between 80 and 160. The generate particles setting in the boundary tab will also help the simulation appear as realistic as possible and, with the right material settings, you'll have incredibly realistic looking fluids for your renders. The example file this month, available in the Dev.Mag website's content section, contains everything I used to create the render displayed here. You will, however, need to bake the simulation yourself; the raw simulation files are far too large for us to upload on our site. Be warned, however; on the settings I've used, render and baking times are rather significant.





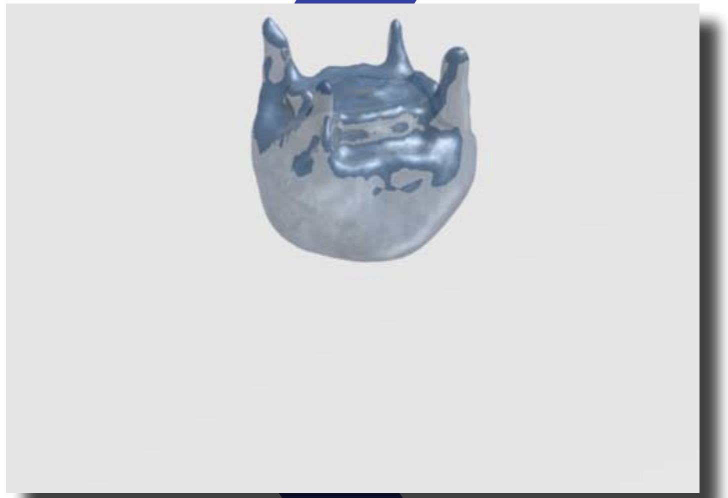


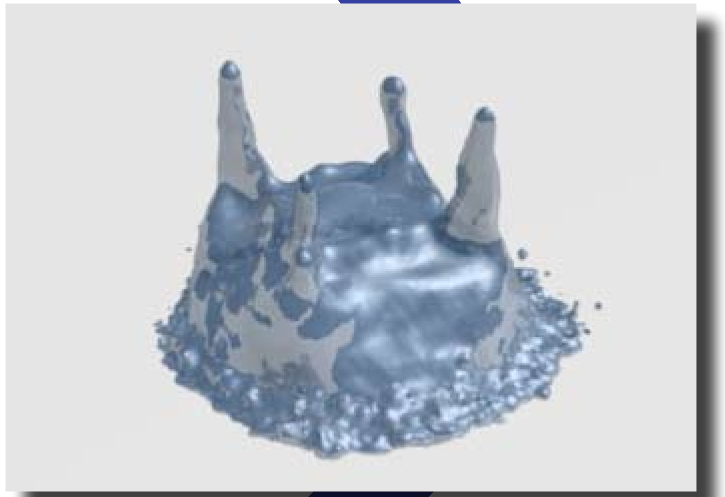


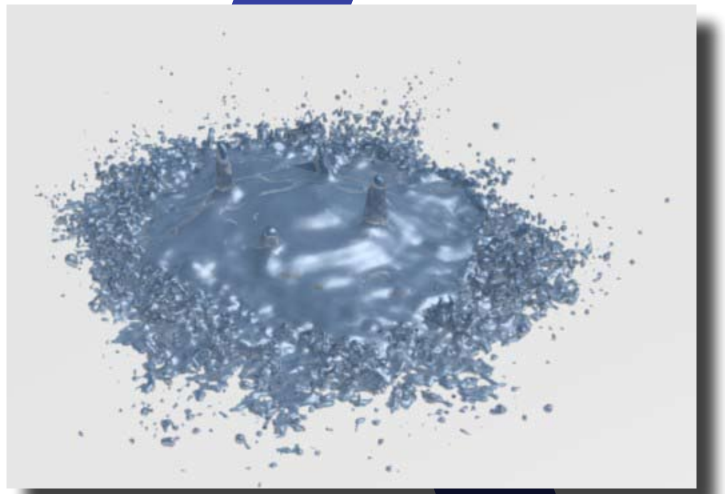


Performance

Fluid simulations are slow. Adjustment of practically every setting on the domain object affects baking time. Resolution and simulation frame count has the largest negative impact, closely followed by the particle generation and surface subdivision settings. It's best, in larger scenes, to test smaller sections of your simulation at a time (if possible), as well as doing so on lower settings. Only ramp up the resolution settings for the final render, and then be prepared to wait quite some time for it to complete.







Rodain "Nandrew" Joubert

THE STUFF OF Dreams

Last year, Microsoft brought Dream-Build-Play into the world – a mewling babe of a game development contest which enjoyed great popularity and a host of high-quality entries. Now, its second incarnation promises to deliver just as much fun, using the XNA 2.0 game development framework to create gems of gaming wizardry for the Xbox 360 console.

"MICROSOFT HAS BROUGHT ABOUT A COMPETITION WHICH PROVES TO BE BENEFICIAL FOR ALL INVOLVED."

As you read this, bunches of eager game developers all over the world are huddling around their systems and scribbling out designs, artwork and code for their very own DBP entries. They work in earnest; with a US$75 000 prize pool and the potential for exposure on the Xbox LIVE marketplace, there's a lot to gain by putting forward the winning entry.

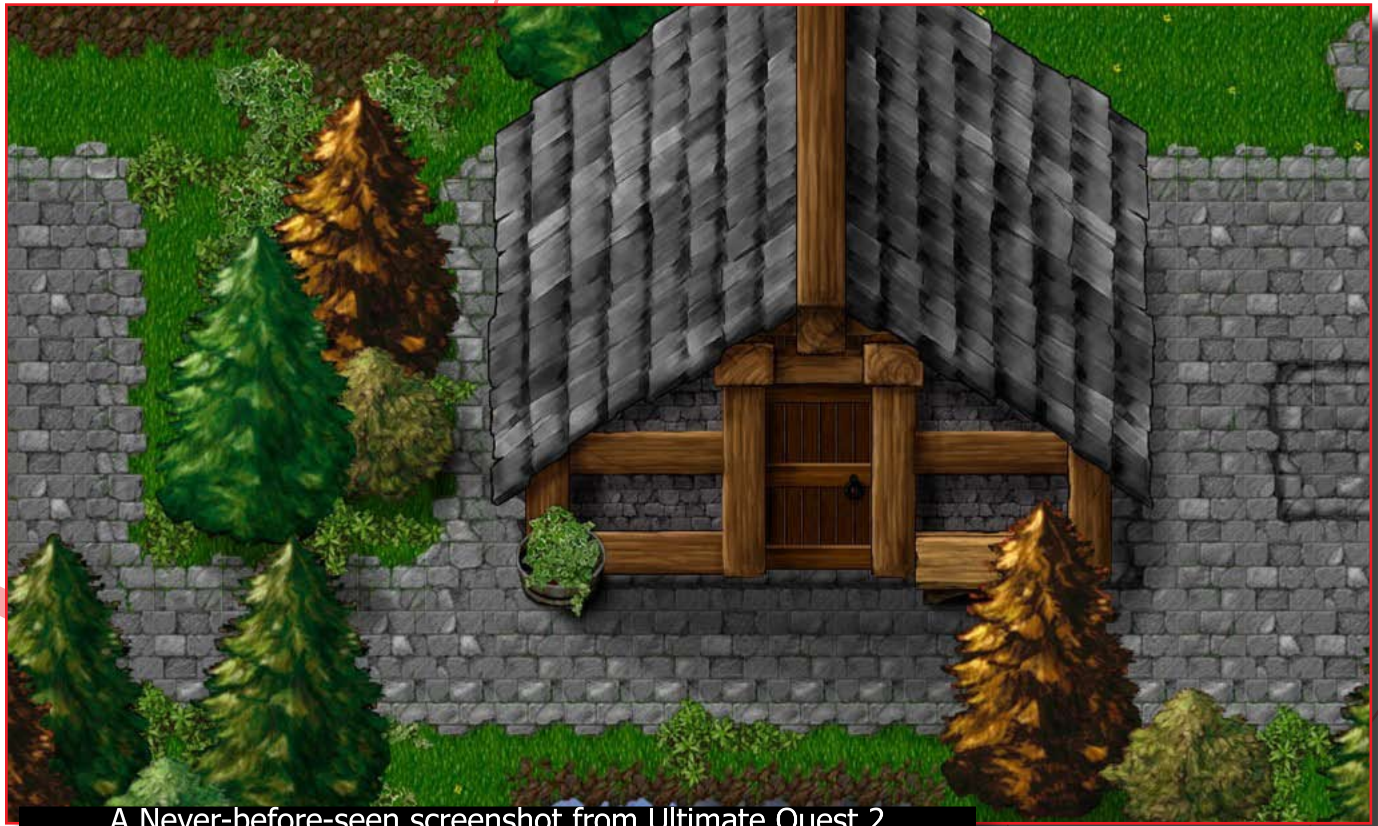
The scope of the competition is considerable, and it produces some very real results. The first DBP contest garnered more than 4500 entries, four of which were actually offered LIVE contracts for mainstream distribution. If other game development contests serve as any sort of testament, it is likely that even more hopefuls will be entering this time around to get a shot at the same opportunity.

So far, each DBP competition has consisted of two main phases: a warm-up challenge and a main challenge. The warm-up is self-explanatory: it's a 'practice round' of sorts to allow entrants to get a feel for the DBP environment and have a go at making a game before the proper competition starts. The really nice thing about these warm-ups, though, is the fact that entrants have the opportunity to win expo passes, internships and Creator's Club subscriptions if their efforts find favour with the judges. This year, warm-up games focused around AI as a core value of their gameplay – winners ranged from hive-mind controllers to sheepdog simulators.

A notable trait in Dream-Build-Play which is not always present in other high-profile competitions is its stress on the fun factor and innovation for the judging process. Production value counts for a humble thirty percent of an entrant's final score, leaving the rest to criteria such as ingenuity and the eternally sought-after desire to continue playing.

Microsoft has brought about a competition which proves to be beneficial for all involved. Individual developers get an opportunity for money, exposure and game development glory. The rest of us reap the benefits of the high-quality indie games from Xbox LIVE as a direct result of the competition, and one of the biggest names in the console business gets to scout for those hidden gems which are scattered throughout the world of low-profile developers.

"THERE'S A LOT TO GAIN BY PUTTING FORWARD THE WINNING ENTRY."


A Never-before-seen screenshot from Ultimate Quest 2

For those who have entered, remember that entries for Dream-Build-Play close on 23 September! Winners are scheduled to be announced in October. For more information on the DreamBuildPlay competition visit http://www.dreambuildplay.com/ , or see a showcase of the warm-up challenge winners available at http://www.dreambuildplay.com/main/winners.aspx

Also keep an eye out for the two Game.Dev entries, **Ultimate Quest II** and **SpaceHack**. Good luck to those who have entered, and to those who haven't – hopefully you'll change your mind next year!

GEAR COUNT:

HEY GUYS! I DIVIDED THE GEAR COUNT BY 0 AND IT WOR- OH SHI-!

ONLINE DEV.MAG

CREATE • DEVELOP • EXPERIENCE

WWW.DEVMAG.ORG.ZA