Panda3D

Game Development

Game development for everyone

A complete Tutorial about game creation with Panda3D and other free software

Fireclaw the Fox

{GrimFang}
open source game studio

Content

Copyright

Version 16.08

Introduction

This tutorial will lead you through every part of the game creation process. It will show you how to start from scratch with a simple idea, write the code, create content, and finally pack everything up into an executable for release to the public. It also tries to be generic and give you as many details as possible, giving the knowledge you need through simple and short chapters, while allowing you to jump right into the things you really want: creating games.

Before we start with the tutorial though, I'd like to make a few things clear. First of all, this tutorial is written for new and intermediate users of Panda3D. It will be necessary to know at least the basics of the Python programming language to be able to understand everything in this tutorial, even though it will also use code snippets from the Panda3D code collection repository found on launchpad.net. As this tutorial will describe every step of the game creation process, it will also go to some length to describe non-programming subjects such as artwork creation. For a more detailed description of the subjects covered in this tutorial, we will give you some links and keywords that you can look for, as well as links to our own more artwork-specific tutorials.

This tutorial also comes with all of the assets that are mentioned in it, so you will still be able to do everything in this tutorial even if you don't have artistic or writing skills. This works in both directions: you can use the sample code from this tutorial and replace the assets with your own and use the hints, tips and tricks mentioned here to make your assets ready to be used with Panda3D.

Note to artists: if you are an artist, you can still use most of this tutorial. You can replace all the assets that come with this tutorial and simply use the application which comes with it with your own graphics, sounds and other artwork. This tutorial will describe where to put your assets as well as what they should contain. You can also follow the asset-specific tutorials which will come at a later time.

Chapter 1: Imagine

At the beginning of every good game is a good idea. As such, this section will be about finding ideas and tracking them for future use.

We will also describe how to enhance a basic idea as well as start the design process and gather as much information for your game as possible.

This part is the most general part of the tutorial and therefore applies to every type of game, including board and card games, as well as game engines other than Panda3D.

Finding ideas

Ideas come and go in the blink of an eye. They can come at absolutely any moment in life, so make sure you always have something to write or draw your idea down. Having that always with you, make sure to go through the world and use all your senses to find new ideas, as they can be found absolutely anywhere. Over the years you should have collected a bunch of ideas that can be used in various kinds of games and other projects.

Also, make sure to play many games and take a note of how professional game developers have done things. If possible, learn from their mistakes, so that you can prevent making them yourself. There will still be plenty of opportunities for you to make your own mistakes to learn from! There are also a lot of sources on the Internet that you can browse through which discuss and study the techniques used in other games.

Even if you want to make a remake of a game, you could your own personal touch to it. You could do this by improving on the existing design or by adding new things that the player can find over time. Making remakes can be a great learning experience, as you know exactly how things should work. By doing so, you get an intimate understanding of the workings of the game, which will come in very useful in future projects of your very own.

The basic idea

After finding several ideas, you should pick a few of them that you can imagine making a game from. You can even pick ideas that don’t seem to work together at first glance if you want to give yourself a challenge creating the game.

Having done that, you should start to narrow down the basics of your game: things like genre, player count, story overview, and so on.

For this tutorial we will use the following idea to create a game:

“A two-player fantasy beat-em-up, taking place in a world inspired by the old Japan, with creatures fighting for the big prize”

So, let's break this idea down into parts and go through them one by one, starting with the first part:

“A two-player fantasy beat-em-up”

In this sentence there are already three pieces of information:

1. The game will be a two-player game.
This means we need to think about we can make the game be played by two people. For this tutorial we’ll make a multiplayer-only game to avoid the need to implement sophisticated artificial intelligence, even though it could be extended with AI at a later point. Also, we’ll say that it'll be a local-only game so we don't need to get into building in networking support yet. As with the AI, networking can be added later on.

2. It'll be fantasy-themed
This gives us more freedom to choose which elements to incorporate into the story and setting, and doesn’t constrain us to the bounds of real-world accuracy.

3. Its genre will be a beat-em-up
Knowing the genre, we can get inspiration from other games that fall into this genre.

The next part is

“taking place in a world inspired by the old Japan”

This part describes the level style and visual appearance of the game. It describes that it will take place in an old Japan-themed world, so we try and look for information about old Japan, such as the architectural and fashion styles. Remember that we don't need to be completely accurate as we are merely building a fantasy world inspired by it, rather than making an exact copy of that time. We can even take elements from different periods in Japanese history.

The last part of the idea describes the following

“creatures fighting for the big prize”

This sentence tells us that the fighters will be fantasy creatures. For example, we can take anthropomorphic animals as well as come up with completely new ideas here.

The other part of this sentence describes their motivation for attending the fights, which is very important to keep in mind when writing a story for it.

Fine-tuning the idea

As soon as you have the basic ideas written down, we can continue to fine-tune the ideas by extending them and removing parts that are too much.

This part is important as it will help you by finding the right balance between the things you want and the things you can do within a reasonable time. After all, you want to play the game someday, don't you? The amount of time you can put into the game depends on how big the team is, or if you're just a single developer, how much knowledge you have and how fast you can work.

To get a good feel for this, you should take part in already existing game projects, as well as game jams or contests like the PyWeek. You could also make a small project of your own with a really short time frame, such as a week or two. This way you'll learn much about both your own skills and game development in general. After a few projects—no matter if they were successful or not—you should know how fast you can work and what you'll be able to get together within a given time. This will help you make better games as well as finish them at the end.

Another good way to fine-tune ideas is the so called Disney-Method, where you and possibly your team members get into various thinking styles. These styles are the Outsiders (the neutral start and end style), Dreamers, Realizers and Critics. For a more detailed description of what you have to do for each style and how to use this method, take a look at this Wikipedia article: https://en.wikipedia.org/wiki/Disney_method.

For this tutorial we have the following enhancements and reductions for our ideas to be able to get the tutorial done within about a week.

1. It should be a really simple game, only very few playable characters (2-3 should do)
2. There should also be only one or two areas to play in
3. We won't have networking and AI in the game
4. The game will have some nice graphical effects to get a nice look

Note that you shouldn't throw away the ideas you have taken away from the bigger idea. Store these ideas somewhere instead, like a growing file folder with all your ideas in it, as you may want to revisit them later.

Splitting up the idea

If we have all our ideas together and tuned them to our liking, we will split them up into smaller parts that we can work on separately. Those steps can describe a single work day or development steps like creating a character, building the menu, connecting all the things together. You can also mix up both which we will also do for this Tutorial.

We know that we want to finish the game within a week, so we have 7 days to plan for and if we have done that, we can further split up the days in different development steps. For the tutorial this could look like this:

Day 1:

Create the idea of the game write down all informations and do the basics for the application like directory structure and copying some starter files from code collections.

This all is rather compact so it doesn't really need to be split any further.

Day 2:

Starting on the character development (models and basic player setup)

Here we can split up to the following steps:

1. Creating the player models
2. Loading the player models into the game
3. Developing the character functionality for one player

Day 3:

Adding further character functionality and environment

This part can be split up as follow:

1. Adding a second player
2. basic environment
3. Implement camera positioning with distance checks to both characters

Day 4:

Adding menus basic options and other GUI elements for the game

Here we have the following parts:

1. Building the main menu using P3Ds internal GUI system
2. Implementing K.O. handling
3. Adding the options menu with simple changeable options

4. Add a player and level selection screen

Day 5:

game enhancements

1. Adding some audio
2. Adding particles
3. Enabling shaders and adding some appropriate ones

Day 6:

Debugging and performance enhancements

Day 7

Releasing the game

The last day can be split in this parts:

1. Writing a manual and creating other game release content
2. packing everything up in an executable
3. Upload the game to a sharing platform or store

That's the plan so far, which this Tutorial will follow. Though, you can also go at your own pace and do as many or less of this points at a day as you like. This only serves as an example of how this project can be structured into days and development steps and how they could be written down. You sure can also extend these points with information you find necessary for your project, especially if you work in a group, the more precise your points are, the simpler it will get for your team members to work on them. If you work on really big projects, you may also find it practical to have multiple plans, one big diagram that shows the projects big plan and then every week create a in depth detailed plan for each day of the week you work on the project. Also make sure to add fun/interesting things wherever possible to each steps, these should let you thrive throughout the project and not getting you bored to soon as you always have something that you can look towards. Note that you should at the beginning of the project put something fun and interesting and towards the end have them at the end of each step, so you have some motivation to start and finish the project at all.

Prototyping

You can take the steps we have above as blueprints for prototypes where each step resemble a separate application that you then can put together to get the big one. Prototyping will also help you focusing on each step if you have problems by getting distracted of other things in your applications to much as in a single prototypes only a single feature and possible subfeatures should be implemented. For example you can have a prototype for the character controls, one for the camera handling and another one for level loading and the menu handling. This way you can also check if the things work by their own and not get disturbed by other game components, so you know it should work if you put it into the complete application.

Using Panda3D and especially python makes creating prototypes really simple as you could even make the most basic Panda3D application with about 4 lines of code. This way, most of the prototypes should also fit in a single file.

Another advantage of prototypes is that you can quickly see if a feature of your application takes more time than you initially expected or maybe is completely impossible the way it was thought to work. Also with prototypes, you can simply create unit tests to make sure the features work even after making some changes to them, but we'll take a deeper look into that at a later section of this tutorial.

Project planning

All the above sections tell you how to go with small scale projects and should be enough for the games a single developer could sure be successfully produce. But on a bigger scale, a project that may take a team several months to years to finish a good project plan is crucial for its finalization. A projects plan contains many parts, we will list them and describe what each of them should contain.

Design document

A Design Document contains the definition and descriptions of all parts of a project, the more detailed the better. Full fledged design documents for applications mostly consist of hundreds of pages describing each of the applications parts in detail, which reach from the applications name over application behavior till pixel perfect definitions of menus and styles. For a game you can say a design document also is a complete walkthrough through the game. For a description of which sections a design document may contain as well as some further descriptions and samples take a look at this wiki page: https://en.wikipedia.org/wiki/Game_design_document

Time and human resource planning

Another big part of projects, especially commercial projects is time and human resource planing. Most of the time you'll have a deadline to hold which is given to you by someone or, if you don't have someone that gives you a deadline, set by yourself. A deadline will definitely help you finish a project as you have something to work up to and may need to make decisions on dropping or shorten parts of the project to finish it in time even if you don't hit the deadline in the end.

For those who know how much time they have every day and especially for teams, you should create a Gantt chart. If you have your project planed as detailed as possible and broken up into “simple” pieces of development, you can take those and put them into a Gantt chart. A Gantt chart consists of many bars representing the time each development step should take and the current state a step should have reached. This way you can also clearly see if a step may takes longer than initially thought and hence can be stretched which will result in stretching the whole time frame of the project.

A good software for creating such charts is “Planner” made by the GNOME project. You can find it at https://wiki.gnome.org/Apps/Planner/Downloads

If you're one of those who can't say how much time you have to work on the project due to other more important things like your job or family, you must use the spare time you have for your project as efficient as possible. You should try to skip all things which wont work for you like for example a detailed time chart as described above. Though, you should always take notes of the current state of your project and have a good overview of what's to be done as you may never know

when you'll have to pause the project for a longer time, and if you come back to a project after a few weeks or even months, you'll be happy about every note you made for yourself to ease and speed up the reentering of the project. But after all, if you don't have much time, you need to scale down the project to something that fits your time frame if you don't want the project to take forever.

Game Design

Now that you know how to plan a project and get a general idea as well as a bit more detailed idea, lets take a look at the game design.

A game design consists of a few parts which we will describe in this section. Basically in a game design, the game's look and feel will be described.

Following we will give a short insight of the parts you can have in the game design process.

Concept art

One of the first things to do, while writing down your ideas, is making as many images as possible visualizing your ideas for the game. Those concept art images should display as much of the game as you can get in them, especially the mood of the game should be visualized in them as this is one of the best ways to show others how the game should look and feel. If your game should get a story, also make sure to draw images for a storyboard alongside creating the story. A storyboard can be of any complexity, some are just like a few scribbles drawn together to show how each scene should be set up, others are like a comic book with really detailed pictures almost displaying how the game should look in the end at that scene. Also some gameplay behavior can be shown with concept art, especially the visualization of specific things like if you have for example special markings to show objects the player can climb up in the game or how enemies should be tracked.

Those concept arts will accompany you throughout the creational process of the game and always are a good thing to look at when implementing features or especially creating the final artwork of the game.

But the most important thing concept art is for, is showing others what you thought, because pictures say more than a thousand words.

Character design

Secondly, one of the most important parts of almost all games is the character design. This contains not just the style, the look of the character, but also it's stats and writing down it's history. The more you know about your characters the better as you can put in many little details in your game regarding the characters and so give them a personal unique touch and your overall game story much more depth. This design process should be done for any character that would get into the game, even though, side characters which may only be seen a few times may not need to get that much detail.

Gameplay

Another part of the game design phase is defining the gameplay. Dependent on the genre of your game, there are already a bunch of gameplay mechanics you can copy from other games of the same genre to make it more easy for the users to get into your game. Humans are creatures of habit hence you should always try to use already set standards wherever possible. Though, to not simply reinvent an already existing game, you should add something new to it and change some behavior if it could be done better or extended.

Creating gameplay you also should take a look at what the fun part of your game should be and try to use it throughout the game and slowly add features to it while the player process through the game to keep the main idea interesting from start to end.

The gameplay design should also contain the necessary information about how the player moves through the levels and the goal of the game.

Also within this section, additional to the general how the player should move throught the game, you should define how the player controls the character and other things in the game, even the menus within it, like if he will use the keyboard and mouse as well as if he will be able to use a gamepad, microphone or other controllers. This should also contain when and how the player can let the character do different actions.

Balancing

One of the more complex parts of a game design is the balancing within a game. Especially for multiplayer games this is a real important part. You should make sure that the game is fair to the player at every point but at the same time challenging to not get boring to fast.

You need to make sure each player has the chance to win whether it be single or multiplayer. In a single player game the enemies or passages or whatever you have in your game needs to grow with the players skills. There are many ways of how the difficulty can be raised over time, like adding more and stronger enemies as well as having more complex tactical parts in it. For a multiplayer session you could take a look at the players level if there are any and group together players of the same level or groups of players that in the end have an equal average skill level, whereby higher ranked players may be possible to drag lower ranked players through a multiplayer session.

Level design

Dependent on your games genre, you will have predefined levels. There are a bunch of level styles to choose from which range from tube styled levels that will guide the player from start to end through the level and leave him just a few ways to go to open world levels where the player can go to wherever he likes at any given time.

Now it's up to you and the style of level you choose as the differ quite much in how they get set up.

Indeed there are still many similarities like how assets get set up to lighting within a level and junk loading if a single level would be to big to be loaded at once.

In the basic level design, you should plan the layout of the maps or open world and the style how the levels, worlds, dungeons and so on should look.

This part is also one of them which should be jointly designed by developers and programmers so the programmers know how to use things in the level and the designers will know what they can even do and how each things should be set up.

GUI design

One of the parts you also have to define in the gameplay is the GUI design. This contains menu styling as well as the HUD and other in game UI designs.

The menu or possibly title screen is the first thing the players will see when they enter a game directly after the logo and preview intro movies. Even if most of the players only tend to stay in the menus for a rather short time, they still have to be designed to be easily browsed through and display the general style of the game to not stand out to much in a bad way. You should also make sure all options in each menu are clearly visible to the user so he doesn't have to search for the start longer than he would play the game otherwise he might even close the game before he even starts it.

The HUD on the other hand should be unobtrusive, intuitive and informative to the player. It should enhance the gameplay but not get into the way of the player for example by hiding things like enemies behind its elements.

As the creation of a UI in Panda3D is a programmers task unless you make a design friendly editor or special language that simplifies GUI creation for designers you should work closely together with the designers. The designers on the other side will still create all the assets and can also create the general design of it using pictures. Those things can then be used by the programmers to set up the UI.

Audio design

The last section of the game design we will talk about is the audio design. The audio design consists of all hearable sounds in your application whether they are music, sound effects, voices or whatever else you may need in your game.

With all that, you'll have enough information to basically start the development of your game. In the next section we will show how to set up the workspace for coders and designers to be able to do the main development of the game.

Chapter 2: Prepare

Now that we have the idea and general planing for our game set, we need to prepare our workspace to be ready to start the development.

This Chapter will describe what you need to do the various jobs in a game development and design process. From writing the source code over art and audio asset creation all the way through until things you may need for the release of the game.

What will be used

We will use Panda3D with python to write our game and blender as our 3D modeling application. Aside of that you should get some image editing software like the Gimp or Krita to create textures and a text editor or IDE like emacs, Geany or Eric to edit your python source code. At the end we will use a vector graphic application like InkScape for icon creation and a desktop publishing application like Scribus for the writing of the games manual.

For this tutorial we will show you how to work with the tools in the following list. If you prefer to work with other ones you need to adapt them by yourself as we can't cover just anything in here, though we try to keep this tutorial as general as possible.

Panda3D - http://www.panda3d.org/

Blender - http://www.blender.org/ with yabee plugin - https://github.com/09th/YABEE

Gimp - http://www.gimp.org/

InkScape - https://inkscape.org/

Scribus - http://www.scribus.net/

TuxGuitar - http://www.tuxguitar.com.ar/

Audacity - http://audacityteam.org/

Due to Panda3D being usable with python it doesn't really matter which operating system you'll be using as long as you can get all the necessary things for compiling the engine if there isn't already a precompiled package available.

As a recommendation you could use UbuntuStudio as operating system for all asset creation as it comes with almost all the necessary software already preinstalled.

The game engine

At first we need to install the Panda3D SDK, you can get the latest stable and development builds here from their official website: http://www.panda3d.org/download.php?sdk Just download the respective package for your operating system and install it. The current stable release as of time of this writing is 1.9.1.

If there's no package for your system available you can also download the source from github https://github.com/panda3d/panda3d and build it by yourself. We won't describe how to do that in this tutorial but you can find the necessary informations on the linked github page.

Even as an artist you should have the SDK installed as it comes with a few tools that will help you check your models if they look good if loaded from the engine as well as some enhancement tools to prepare your models for the game.

You should also get the runtime of the engine, which will install the necessary things to run packed executables (namely .p3d files) created by the engine. The “runtime” will also be installed on all client machines that will run your game. You can simply get and install the runtime from here www.panda3d.org/get/

The graphics pipeline

After you have installed the SDK, you can continue with installing blender. If you're on a Linux system with a software manager you should be able to install it using that, otherwise you can get the latest blender version from their website: http://www.blender.org/download/ Like with the Panda3D SDK, simply download and install it on your system. Additional to blender we also need the exporter plugin to be able to export the models build with blender to Panda3D's own model format called “egg”. You can get the latest version and information of this plugin from here: https://github.com/09th/YABEE To install the plugin, just follow the simple instructions given in the “YABEE_HowTo” document in the “Installation” section.

Having that, you'll be able to create models and export them in Panda3D's model format. Tho, as you may know, you also need something to create and edit textures. This can be done using the Gimp. If you use a Linux distribution, this application may already be installed or at least be available from the software repository on your system. Otherwise, just download it from the Gimp website and install the package fitting to your system. Informations how to get and install the Gimp can be found here http://www.gimp.org/downloads/.

The audio production

Each game needs audio to be really complete, hence we will have a few audio production software alongside all our other software to create and record some simple audio files. The first one is as mostly you won't have a full live orchestra or band to create some game music for you, TuxGuitar which we will use to create some music and maybe sound effects for the game. To get better music from it you should think about getting some audio effect software like some LADSPA Plugins you can connect to TuxGuitar via the jack audio server and better sound fonts like the ones you can simply get for timidity. To record audio whether it's from TuxGuitar or some live microphone source you can use tools like audacity or for the ones with more professional needs in recording, Ardour.

Whatever you use to create audio, in the end you should be able to output your audio files as ogg files as those would be small in size and yet have good quality. The even are useful for streaming if you build networking into your game with asset streaming. During the creational/design phase of the audio production, you should use uncompressed audio to not loose any quality while editing the audio. Hereby wave and flac audio files would be best suited.

For the release

If you plan to release your game, you should have some good content coming with it aside of the game itself. This can be a nice manual as well as application icons and videos showcasing the gameplay etc. So you should additionally have the following software or similar ones installed on your system to create the extra content for the release.

First, scribus. This desktop publishing application is really useful for manual writing, especially if you want to print your manual in the end for hardcopies of your game. As with the Gimp this application should be in the software repository of your most favorite Linux distribution as well as being available from their website with install instructions found here http://www.scribus.net/downloads/stable-branch/

Also don't miss an application for icon creation. For this task, inkscape is really capable. It is a vector drawing application and hence fits perfect for drawing icons which then can be simply scaled to various sizes as mostly you need icons from 16x16 pixels to up to 265x265 pixels or also 512px big for the high resolution screens. Inkscape again can be installed via the software repository of your Linux distribution as well as being downloaded here https://inkscape.org/en/download/

Having that done you'll be ready to create awesome games and can proceed to the next section.

Chapter 3: Develop

With a prepared workspace and a good game idea we can finally start the coding of it. So we will start the project by creating a directory structure, some python files and some other files useful for the development.

Having a good structured project is a really important thing if a project grows. It helps you and other possible co-developer find things faster which in the end reduces the production time and frustration if you may come back to a project after a while. A good rule of thumb might be to have a relative flat (2-4 directories deep) and if necessary wide directory and file structure. Also make sure to not put all the files in just one directory, better create a few general directories like core, networking, UI and so on whit a few files in them. If there's just one file in a directory, you should think of putting it into some other folder together with some other files or simply move the file one folder up.

For source files a good rule of thumb is to have them rather short. For example if a file exceeds 500 lines of code you should think of splitting it up into multiple files and/or refactoring it. Also make sure to not mix up multiple coding styles in a project, otherwise this will just hinder you from reading through the files and finding things fast as well as further coding in the project. Which coding stile you use is up to you.

That's all for good coding practices for now, from here on we'll continue with creating the project for this tutorial.

Preparing the structures

At first we create the basic structure for our project. Choose a base directory to start with, this can be something like your documents or home folder or a dedicated workspace folder. In there create a directory with the name of your project, if you don't know how you'll call your project just yet, you can also use a temporary name like “myGameProject1” or something more specific to the game for example “SimpleRTSGame” or “JumpAndRunGame” how you want to call it is up to you as long as you know what's behind the folder name.

For this tutorial we will create a directory called “P3D Tutorial”. Within this directory create two folders and name them “design” and “game”. The design folder will be used to store all our source files for the assets like blender model and gimp image files. The game directory will contain our python source files and an final assets for the game.

The last folder we will create for now is called “assets” and is located in our “game” directory. This will contain the final assets of the game which are egg files, textures and particles.

Within the assets folder you can also already create two other directories, one called “characters” and the other called “levels”. These will respectively be used to store the character models and textures and the selectable level models usable in the game.

All the other folders which we need will be created later during the development as soon as they become necessary. So for now we have the following structure:

/P3D Tutorial/
/P3D Tutorial/design
/P3D Tutorial/game
/P3D Tutorial/game/assets
/P3D Tutorial/game/assets/characters
/P3D Tutorial/game/assets/levels

Creating a simple P3D application

Now that we have a base for our project we can finally get some code to start a minimalistic Panda3D application to see if everything is working. Most time the base of a Panda3D application is the same, you load and set some configurations prepare your game content and at the end start the engines main loop. Therefor you can safely use some already made code snippets to get you started faster. These snippets and sample codes can be found at various places on the web, most of them in the Panda3D forums which are also a good place to ask questions if you have any troubles with the engine. On collection of such snippets, made by the author of this tutorial, can also be found at Launchpad: https://launchpad.net/panda3dcodecollection For this tutorial we will also use some parts of this collection, so either check it out locally or just download the parts you need at that time.

The first part of code will always be the main.py script so copy the “template for main.py” script into your …/P3D Tutorial/game/ folder and rename it to main.py.

Before we start the script for the first time, we need to change some things in this file to fit our projects details.

First search for the line __builtin__.companyName = “Your Companies Name” and change the string to the name of your company. If you don't have any company or no name for it yet, you may enter your name or anything else that you want to have. Note this variable must be set as it will be used as the top level for your games directory structure in the your systems home folder.

Search for the line __builtin__.appName = "Game Name" and change Game Name to something more fitting. For this tutorial we will use Japanese words for Battle Creatures, 戦いの生き物 as this will fit quite good for our games genre and setting. Note as we shouldn't have non Latin alphabet in our code as we never knew who will take a look at our code and if he has the correct encoding and fonts installed, we will add “Tatakai no ikimono” for the appName. The application name will be used for a few things. It will be used for the name of the directory which will be created within your system users home folder. This directory is used to store game related content but more of that later. The name will also be used as window title.

Another variable you should always keep updated is the __builtin__.versionstring which represents a string containing the applications version number. The current version style of it is YY.MM where YY is the year and MM is the month with leading 0, but you can change this to any format you want, for example if you prefer the 1.0v style or maybe even use the words alpha or beta. If you have an automated build procedure, you may also want to have this set programmatically. But that's all up to you.

The application is set up to run in fullscreen mode and can be quit at any time by hitting the escape key. So if you now run that script from the terminal using python main.py you should notice a

window poping up which then will switch to fullscreen mode, so you see a black screen and your mouse cursor. If you close the application with the escape key now and check in your home folder, you should see a directory called after the name you have entered for the appName variable. If that's the case you have done everything correct so far. In that folder you can find log files and a file ending with .prc. This file is a configuration file, if you open it with a text editor you can set a few configurations like change fullscreen to #f so the application gets started in windowed mode which might be better for development sometimes. The other variables in that file should be quite self explanatory. Keep in mind that user edits to this file will be preserved when the file gets rewritten by the application and the configurations haven't changed during the applications runtime. If you want the default configurations again, simply delete or rename the prc file and it will be rewritten at next application start.

Make a character

Creating the code for a game is one important part. Though, now we get to another part which is also a real important one. Creating game content. Without 3D models and textures a game wouldn't be the same and 3D games couldn't even exist to the extent they do now. So let's begin by making a character for the player to control and import it into the game.

If you're not an artist but a coder, the Creating characters section may not be of much use for you and you should just skip to the Bring the character into the game section. Otherwise, the next section will have a few important notes for you regarding model generation in general and character creation specific as well as information about how you will be able to create and export models so they will be usable with the Panda3D engine. Also it will tell you some information about what you can and should do to help the coders developing the game.

Creating characters

We won't get that much into detail on how to create characters with blender in this tutorial, therefor many good tutorials already exist and we would just copy them in the end. Also if you say, you don't have art skills and just want to focus on coding, you can use already made models which can be found on a bunch of websites like http://opengameart.org/. So we only show you some tips and tricks for how to make them game ready with Blender and usable in a Panda3D game.

A good source of blender tutorials is the blenderguru.com website as well as the blender noob to pro wikibook which can be found here: https://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro

And for a good tutorial on character rigging take a look at
http://www.blenderguru.com/tutorials/introduction-to-rigging/

Also there are tools that can help you simplify the creation of human characters. One of these tools is MakeHuman: http://www.makehuman.org/

Here are now a few things that you can use if you create your own characters for games.

1. Make sure the origin of the character is at it's very bottom. This will greatly simplify the correct placing of it in the world.
2. Keep the polycount low, even if current graphics cards can handle a bunch of polygons already, you should use other better suited tricks to enhance the realism and look of your characters.
    1. Having good textures for your characters fitting the style you want for the game
    2. Use specialized shaders like bump-map/normal-map and ambient occlusion. The textures you need for those can be generated from higher polygon models that have been created with sculpting tools and then baked on a lower polygon model. The keyword you can search for on this topic is retopo (“remake topology”). There are several ways on doing that reaching from handmade to automatic tools.
        1. If possible, use real existing objects for your high polygon models and textures. These can be for example be scanned with 3D-Scanners
3. Put as many objects as possible together into one. For example if the characters clothes shouldn't be changeable, don't waste your precious resources creating the bodyparts below the clothes and directly model the clothes at those parts.
4. Keep an eye on the objects name for example don't keep the default names like cube or bone.001, especially if it doesn't even resemble the object stated in the name anymore. Better use descriptive names and also keep some naming conventions like having a col_ prefix for all collision solids or .L .R postfix to differentiate between left and right objects/bones/etc.
5. Define a real world value for the 3D unit or set it to metric units if possible and don't mix up to many different scales. You can for example say all level models have a scale of 1 and characters are build at a scale of 2 to ease the modeling of details on them if necessary.
6. In blender, use Actions for animations. These can simply be exported by the yabee plugin.
7. Also for animations, use motion capturing and rotoscope the animation from real world movements.
8. If you can't afford highly detailed animations, you can also create animations with only a few main key frames in the most extreme poses of an animation. Those can then be automatically be smoothed and procedurally controlled with IKs within the application.
9. Use a version control system for your files this will let you move back to earlier versions of the files whenever you break or messed something up in them.

This tutorial comes with two character models in the .blend and .egg file format that you can use for the game.

If you want to export the egg file from the blend file by yourself, you need to follow these steps.

1. Deselect everything and then select only the low polygon character model and the armature
2. Make sure in the Dope Sheet (Action Editor Mode) the current action is unlinked and all actions have a fake user. Otherwise yabee will export the current animation twice with a . 001.egg extension.
3. Go to File → Export → Panda3D (.egg)
4. Enter a name for the exported file
5. make sure that the following checkboxes on the left pane are checked
    1. “Copy texture files” and the “Tex. path” is set appropriate
    2. “All actions as animations”
6. Hit “Export to Panda3D EGG” on the top right.
7. After the model and animations have been exported you can copy and rename them to fit your need. (For this tutorial you should name the main model to char.egg and from the others you should remove the character#- prefix. This is also described below again.)

With all that information you should now be able to create character models for this tutorial game and other games you like to createn.

From now on we will start with the coding work that has to be done to get the character into the game.

Bring the characters into the game

Now that we have a character for our game, we need to bring it into our game. To do that, copy the characters egg and texture (.png) files into a sub folder of the assets/characters directory called “character1” and rename the main model file to char.egg and the animations to the lowercase name with only the animation description like “idle.egg”, “walk.egg” or “kick_r.egg”. Having them now at the right place, we can write the code necessary for loading them into the game.

Basic setup

As we won't load the characters directly in our main.py script we first create another file where we store our player parts in. In the game directory create an empty file and name it player.py. This will be used to load our character, handle user input and store some values like health for the character.

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

# Panda3D imoprts
from direct.actor.Actor import Actor

class Player():
    def __init__(self, charId, charNr):
        self.charId = charId
        charPath = "characters/character%d/" % charNr
        self.character = Actor(
            charPath + "char", {
                "Idle":charPath + "idle",
                "walk":charPath + "walk",
                "punch_l":charPath + "punch_l",
                "punch_r":charPath + "punch_r",
                "kick_l":charPath + "kick_l",
                "kick_r":charPath + "kick_r",
                "Hit":charPath + "hit",
                "defend":charPath + "defend"
            })
        self.character.setH(90)
        self.character.reparentTo(render)
        self.character.hide()

    def start(self, startPos):
        self.character.setPos(startPos)
        self.character.show()

    def stop(self):
        self.character.hide()
```

Let's get through this code and see what's in it.

The first few lines simply describes it as a python script, name an author and give it a license.

Nothing special so far. Then we have a import statement of one of the game engines classes. We will talk about that class in a bit. The next few lines are also quite standard python so I will just give a brief description of them. Starting with this line

```
def __init__(self, charId, charNr):
```

We call the init function with a character number so we can use the class for any character model that we have stored in our assets and not need to write a class for each character as well as a character ID which we will use later for the collision detections.

```
charPath = "characters/character%d/" % charNr
```

using a variable we will shorten the next few code lines where we will need the path to our character. Also we use the character number as stored in charNr to choose the folder of the players selected character.

Now we get to the Panda3D specific functionality.

```
self.character = Actor(
    charPath + "char", {
        "Idle":charPath + "idle",
        "Walk":charPath + "walk",
        "Walk_back":charPath + "walk_back",
        "Punch_l":charPath + "punch_l",
        "Punch_r":charPath + "punch_r",
        "Kick_l":charPath + "kick_l",
        "Kick_r":charPath + "kick_r",
        "Defend":charPath + "defend",
        "Hit":charPath + "hit",
        "Defeated":charPath + "defeated"
    })
```

This function call tells the Panda3D engine to load an animated model. Actor hereby is a global function set up by the engine. The first argument of this function is the main model of our character and the second is a map of name and path strings. The names will be used to later chose which animation should be played and the other string determines the path to an egg file that contains an animation in it belonging to the main model. Note, that you don't need and also shouldn't add the .egg extension to the paths as later on the egg files will be converted into the bam file format which we will describe in a later section.

```
self.character.setH(90)
```

This line simply rotates the character by 90 degrees around it's upward pointing axis which is the Z-axis in this case. There are a few more functions like that but more of that later.

```
self.character.reparentTo(render)
```

```
self.character.hide()
```

With this two lines of code we first tell the engine it should add the character to the render node. This has to be done for any object that should be rendered in the 3D-space created by the engine. Whereas render is a global variable set up by the engine. Directly after that we call the hide function as we don't want the character to be shown directly after it got loaded.

Within our start function we currently have just two lines,

```
self.character.setPos(startPos)
```

and

```
self.character.show()
```

the first one sets the position of our character in the 3D-space to the one given in the startPos variable. The next line let the character appear in your application and hence is the opposite function to our previously called hide function.

Respectively to the start function, we also have a stop function which we will use whenever we want to completely stop the player, for example if we leave the game later to the menu screen or if we would have a pause screen. In the stop method we currently just hide the players model.

Now that we have our basic player class ready, we need to instantiate it within the main class to load it and get the player in the game. To do this, we first need to import the class into our main script, so search for the line

```
#TODO: Put your game imports here
```

and replace it with

```
from player import Player
```

This statement is important to be able to use the player class in our main script. Without that, the main script won't know where to find the class.

next search for

```
#TODO: put game content initialization stuff here
```

and replace that line with

```
self.player = Player(0, 1)
```

Here we have instantiated our player, which is the first step to get it visible in our game and we also

tell it, that we want the first character model to be loaded. Later on we will do that for every player in the game, but for now we will have just one which we can play with and test some things.

Finally put the following line in the enterGame function by replacing the “pass” line

```
self.player.start((0, 8, -0.5))
```

This line will start the general player functionality and place the model at the center of the screen by placing it 8 units “into” the screen and 0.5 units down so it will be seen by the camera and hence visibly rendered. Note that the numbers in the inner round brackets represent the x, y and z positions. The values of these numbers represent kind of undefined units within the 3D space created by the Panda3D engine. Though, you can define them for yourself and say, that for example 1 unit equals one meter. Then you can model your 3D models according to that scale.

Now that we have that done and start our game, we will already see our character, which is nice, tho nothing really to special. So let's move on to something more. In the next step we will set up a Finite State Machine or FSM for short and play animations of the character in the various states of this FSM. For a detailed description of what a FSM is and what it is good for, take a look at this wikipedia article: https://en.wikipedia.org/wiki/Finite-state_machine For our player class, we will use the state machine to handle the various actions a player can do for example idling, walking, attacks and it's death state.

Simple animations

Let's see how we can implement a FSM for our player class. First of all, we need to import the FSM class from the P3D engine. Just add the following line above the player class definition in the player.py script.

```
from direct.fsm.FSM import FSM
```

This will let you use the FSM class within the player script. The next part you need to do is to let the Player class inherit the FSM class, which can be achieved by adding FSM into the brackets after “class Player” so it look as follow.

```
class Player(FSM):
```

another thing you need to do to be able to use the FSM in this class is to initialize it by adding the following line at the top of the Player class's __init__ function.

```
FSM.__init__(self, "FSM-Player%d"%charNr)
```

after that you'll be able to use the Player class as an FSM and you can implement state functions in it. These state functions are the enter and exit parts of a state and hence the names always start with either of them for example enterMainState and exitMainState. There are no specialized functions

for the time while the FSM is in a current state, but you can always check in which state the FSM currently is. Also note, that you don't have to implement a enter and exit function to a state and can even ignore them if you don't need them at all.

Before we implement some of those state functions let's take a look at animating our character. As we already have loaded our animation files into our character variable through the actor interface, we can simply call functions like play and loop with the name of an animation to start it.

With this information, we can write our first state function which will be enterIdle. This function will then be called whenever the FSM of the Player class moves into a state called Idle. Note that the states are case sensitive. The following code shows how this function is written. It doesn't matter where this function is located within the Player class but for now we just add it to the bottom of the class.

```
def enterIdle(self):
    self.character.loop("Idle")
```

Now that we have a enter function to the Idle state, we also need to change to that state as a FSM's state isn't set until you explicitly set it for the first time. To do that we can call the “request” function which got inherited into the Player class from the FSM class.

For now, the best way to change to the Idle state is in the start function of the Player class as the first state should be idling. To achieve this, add the following line at the end of the start function within the Player class.

```
self.request("Idle")
```

After that you should see the character animated with the idle animation if you start the application again.

User interactions

Now that we have a first animation playing, we can go and bring in some user interactions into the game. There are several ways to get the user input and work with it, for example you can use Pandas event system to catch keyboard input as well as a Polling interface and even raw keyboard events. For this tutorial we will use the Polling interface within a task function. Tasks are functions which get called every frame as long as they are active.

Knowing this, we are able to add keyboard interactions to our character. We will implement that directly in our Player class for now using a set of predefined movement variables which we can later simply extend for two player (on one Machine) support. Back at our Player class, add the following line inside the __init__ method.

```
self.walkSpeed = 2.0 # units per second
self.leftButton = KeyboardButton.asciiKey('d')
```

```
self.rightButton = KeyboardButton.asciiKey('f')
```

This variables define how fast the character is moving from left to right as well as the key mapping. We use the 'D' and 'F' key for letting the character moving left and right respectively. I've chosen these keys as they are next to each other and also will leave enough space for the other necessary buttons for the player control as well as enough place for the buttons for player two. The rest of the buttons will be added at a later section. Having those, we will add the function in which we will use these variables.

Add the following lines somewhere in the Player class

```
    def moveTask(self, task):
        speed = 0.0
        isDown = base.mouseWatcherNode.isButtonDown
        if isDown(self.leftButton):
            speed += self.walkSpeed
        if isDown(self.rightButton):
            speed -= self.walkSpeed
        yDelta = speed * globalClock.getDt()
        self.character.setY(self.character, yDelta)
        return task.cont
```

This is our main worker function of the Player class. It will run as long as the player is active and handle all the movements the player can do. Currently it is only able to move the player left and right on the x-axis without restrictions, so you can also move out of the screen. Let's take a closer look at the lines of that code snippets.

In the first line, the function definition we await an extra argument called task. This is always the case for task functions. This task variable will be of type direct.task.Task and contains a few important things for task handling.

Next we have a variable definition which we will later use for setting the movement speed of our character. Following that we store the function to check for keyboard keys in a shorter named variable. Note the name may be a bit misleading but in Panda3D the mouseWatcherNode also handles keyboard input. After having set up those, we will use that function to check for our previously defined buttons and set the speed respectively.

After we know which direction to move we calculate the delta movement speed which is just our movement speed multiplied with the delta time of the frame, meaning the elapsed time of that frame in seconds. Note that globalClock is a global variable which can be used anywhere in your application and is mostly important for use in tasks.

Having all that done, we just need to apply the delta movement on the x axis to our character which we will do with the setY function. There are a few of such functions like setX, setZ and for angular rotations setH, setP and setR which relates to heading, pinch and roll. In this example, you also see that we pass the character as first argument to the setY function. This will set the new y value of the

character respective to the character itself, so that it moves always in the characters y direction no matter of it's rotation and also from the characters last position. As we have previously set the characters heading to 90 degrees, it looks to us as that the character is moving on the x-axis of the screen which is currently also the global x-axis but as seen from the character this is his local y-axis.

The last line of that function tells the task manager that it should call this task Function again in the next Frame. This will it do until the task got terminated externally.

Now that we have a task function, we also need to activate that task which can be done by adding the following line at the bottom of the Player class's start function.

```
taskMgr.add(self.moveTask, "move task %d"%self.charId)
```

Having done that, you should be able to move the character left and right with the 'A' and 'D' keys on your keyboard if you restart the application.

Also add this line to the “stop” function of the Player so the task gets stopped whenever we stop the character itself.

```
taskMgr.remove("move task %d"%self.charId)
```

Moving the character is nice, but for now it doesn't look to realistic as the character always stays in the idle state and hence only plays the idle animation which let him appear to be floating over the ground. To fix that, we need to implement a few more states. The following code snippet will show you what to add at the bottom of the Player class.

```
    def exitIdle(self):
        self.character.stop()

    def enterWalk(self):
        self.character.loop("Walk")
    def exitWalk(self):
        self.character.stop()

    def enterWalk_back(self):
        self.character.loop("Walk_back")
    def exitWalk_back(self):
        self.character.stop()
```

As you already know the enter and loop functions we will directly skip to the new things in this small snippet. The exit and stop functions. As mentioned above each state of a FSM can have an enter and exit function. The exit function will be called whenever the state as stated in the function name is left. Within this function we call stop on our character which will stop all animations currently playing on that actor.

Next we will make use of our new states by changing them whenever the actor is moving or stops

moving. This can be achieved quite simple by adding the following lines inside the moveTask function right before the return statement.

```
if speed != 0.0 and self.state != "Walk" and self.state != "Walk_back":
    if speed < 0:
        self.request("Walk")
    else:
        self.request("Walk_back")
elif speed == 0.0 and self.state != "Idle":
    self.request("Idle")
```

Using the speed variable we always know if the character is moving or not as well as the direction in which the character is moving and with the state variable which we got through inheriting from FSM we also know which state the FSM currently is in. Knowing that we can check if the character is walking and not already in the Walk and Walk_back state. If so we will request the walk or walk backward state to play the respective animation. If we would only check for the speed variable, the character would change to the walk state every frame and hence start the animation over and over again, though, as we loop the animation at the entering of that state, we can skip that request if the character is moving and already in that state. Same goes for the Idle state but there only if the character is standing still.

Having done that, you should see your character walking and idling in the game respective to your keyboard input. That should look much better now but the character still misses a bunch of things.

Extended character movements

From here on you should know how to add states and how to change between them, so the following code to add all the other states of the character should be clear in what they do. Just add them below your other enter/exit functions:

```
    def enterPunch_l(self):
        self.character.play("Punch_l")
    def exitPunch_l(self):
        self.character.stop()

    def enterPunch_r(self):
        self.character.play("Punch_r")
    def exitPunch_r(self):
        self.character.stop()

    def enterKick_l(self):
        self.character.play("Kick_l")
    def exitKick_l(self):
        self.character.stop()

    def enterKick_r(self):
        self.character.play("Kick_r")
    def exitKick_r(self):
        self.character.stop()

    def enterDefend(self):
        self.character.play("Defend")
    def exitDefend(self):
        self.character.stop()

    def enterHit(self):
        self.character.play("Hit")
    def exitHit(self):
        self.character.stop()

    def enterDefeated(self):
        self.character.play("Defeated")
    def exitDefeated(self):
        self.character.stop()
```

The only difference we have here to the other functions we previously set up is the play method we call on the character class. This function is very similar to the loop function but instead of looping the animation, it plays the given animation just once and then stop and stay at the last frame of that animation.

With all those animations, we now also need to define when they should be entered and played. As we want to map them all to some keyboard events, we first extend our button variables. So add the following lines of code at the end of the __init__ method.

```
self.punchLButton = KeyboardButton.asciiKey('q')
```

```
self.punchRButton = KeyboardButton.asciiKey('w')
self.kickLButton = KeyboardButton.asciiKey('a')
self.kickRButton = KeyboardButton.asciiKey('s')
self.defendButton = KeyboardButton.asciiKey('e')
```

The next thing we need to change is the moveTask function. In it we will now map all our keyboard keys to the given states and also bring in some logic, so that some animations won't be stopped in the middle of their playing as well as hindering the player from doing to many things one after another.

```
    def moveTask(self, task):
        if self.attackAnimationPlaying(): return task.cont
        speed = 0.0
        isDown = base.mouseWatcherNode.isButtonDown

        if isDown(self.defendButton):
            if self.state != "Defend":
                self.request("Defend")
            return task.cont

        # Check for attack keys
        isAction = False
        if isDown(self.punchLButton):
            isAction = True
            self.request("Punch_l")
        elif isDown(self.punchRButton):
            isAction = True
            self.request("Punch_r")
        elif isDown(self.kickLButton):
            isAction = True
            self.request("Kick_l")
        elif isDown(self.kickRButton):
            isAction = True
            self.request("Kick_r")
        if isAction:
            return task.cont

        if isDown(self.leftButton):
            speed += self.walkSpeed
        if isDown(self.rightButton):
            speed -= self.walkSpeed
        yDelta = speed * globalClock.getDt()
        self.character.setY(self.character, yDelta)
        if speed != 0.0 and self.state != "Walk" and \
                self.state != "Walk_back":
            if speed < 0:
                self.request("Walk")
            else:
                self.request("Walk_back")
        elif speed == 0.0 and self.state != "Idle":
            self.request("Idle")
        return task.cont
```

In this section, the new lines are marked in yellow. Starting with the first line, we will check if the

character already plays an attack animation. We will create that function in a bit, for now it's just important for us to know that if an attack animation is playing we wont let the player do anything else yet hence we return with a task.cont so the function will be called at the next frame again.

Our next block is about the defend state of the character. If he changes to that state the character will play the defend animation and as long as the player presses the defend key he will stay in that state as well as stay at the last frame of the animation if it played through. Again we return with a task.cont here.

The last big block we add are the attack keys. Here we just check which attack key is pressed and enter the respective state. Also we add a check variable called isAction which we always set to true whenever an attack key is pressed. This variable will then be used to check if we leave the function as we don't want to “disturb” the attack animations by playing a possible movement function added afterwards.

Now we will see how the attackAnimationPlaying looks like. After all it's quite simple to check for a few animations if they are currently playing or not by just adding this function somewhere in the Player class.

```
    def attackAnimationPlaying(self):
        actionAnimations = [
            "Punch_l",
            "Punch_r",
            "Kick_l",
            "Kick_r",
            "Hit"]
        if self.character.getCurrentAnim() in actionAnimations: return True
```

All we do here is creating a list variable with all names of the attack animations the player could possibly use and then check it against the value as returned by the getCurrentAnim function which will return a string containing the currently playing animation.

With all these new things you should now be able to walk around with the character and do some attacks. This is quite nice so far, but without some other player and actual fighting gameplay this gets boring rather fast so let's move on to the next section and add another player to the game.

Add two player gameplay

From here one we should have the controls and basics for a single player class. In this section we will bring up the necessary things to add a second character to the game which can be controlled by a second player or at a later state maybe even by an artificial intelligence (AI). Now lets start with extending our already existing Player class so we can use it for multiuser usage.

Basics

The first thing we will change and extend is the init method of the Player class so it will look like the following.

```
    def __init__(self, charId, charNr, controls):
        FSM.__init__(self, "FSM-Player%d"%charNr)
        self.charId = charId
        charPath = "characters/character%d/" % charNr
        self.character = Actor(
            charPath + "char", {
                "Idle":charPath + "idle",
                "Walk":charPath + "walk",
                "Walk_back":charPath + "walk_back",
                "Punch_l":charPath + "punch_l",
                "Punch_r":charPath + "punch_r",
                "Kick_l":charPath + "kick_l",
                "Kick_r":charPath + "kick_r",
                "Defend":charPath + "defend",
                "Hit":charPath + "hit",
                "Defeated":charPath + "defeated"
            }
        )
        self.character.reparentTo(render)
        self.character.hide()
        self.walkSpeed = 2.0 # units per second
        if controls == "p1":
            self.character.setH(90)
            self.leftButton = KeyboardButton.asciiKey('d')
            self.rightButton = KeyboardButton.asciiKey('f')
            self.punchLButton = KeyboardButton.asciiKey('q')
            self.punchRButton = KeyboardButton.asciiKey('w')
            self.kickLButton = KeyboardButton.asciiKey('a')
            self.kickRButton = KeyboardButton.asciiKey('s')
            self.defendButton = KeyboardButton.asciiKey('e')
        elif controls == "p2":
            self.character.setH(-90)
            self.leftButton = KeyboardButton.right()
            self.rightButton = KeyboardButton.left()
            self.punchLButton = KeyboardButton.asciiKey('i')
            self.punchRButton = KeyboardButton.asciiKey('o')
            self.kickLButton = KeyboardButton.asciiKey('k')
            self.kickRButton = KeyboardButton.asciiKey('l')
            self.defendButton = KeyboardButton.asciiKey('p')
```

The yellow marked code shows the new parts. First we add a controls argument to our method. This

will be a string variable where we await a value of something like p# where # must be 1 or 2 for now as we have a two player only game.

The next change is, that we move the previously defined *Button variables as well as the heading change (.setH) of the character within an if-block. This if-block checks whether the controls should be for the first or second player in the game. For the next if-block we add a heading of -90 so the character looks in the opposite direction and hence towards our first character. The buttons we map to our second players character are all on the right side of the keyboard using the left and right arrow keys as well as some ascii keys for the attack and defend actions. Note that we mixed left and right as we have set the heading to -90 but haven't and wont changed the code within the moveTask where we have specifically said that left is - and right is + on the x-axis. Hence moving left results in moving right on the second character and right results in moving left.

Now we need to add and start the second character to our game which we will do in the main script. We do that very similar to our first character, just change the values to fit the needs as a second character. The following lines will show what to add and change in the main.py script.

```
        #
        # initialize game content
        #
        self.player = Player(1, "p1")
        self.player2 = Player(1, "p2")
```

Due to our changes in the init method we also need to add the necessary argument to our first player. The second player variable is set up just like the first one, except that we set the controls to p2 which will result in the second keyboard mapping.

```
    def enterGame(self):
        # main game code should be called here
        self.player.start((-1, 8, -0.5))
        self.player2.start((1, 8, -0.5))
```

Also in this part, the second player is called just like the first one. Here the only differences between them two is the first value which represents the position on the x-axis, we will place the first character to the left and the second character to the right on the screen.

From here on if you start the application, you should see two characters facing each other, which can both be controlled via the defined keyboard buttons. Maybe you notice that both players can also run through each other by now. This will be our next point we will take a look at, namely collision detection.

Advanced

Collision detection will check for two or more objects to interfere with each other. It can be used to prevent objects move through other objects as well as check if an object is touching something to trigger things like door switches or traps. Panda3D comes with a bunch of good professional grade collision and physic systems. Those are Panda3D's own implementation which is mostly good for simple physics and collision detections, ODE, the open dynamic engine which has good implementations on joints and should be good for technical simulations and the last but not least, Bullet, which is a good capable physics and collision engine for general purpose which was already used in big commercial games and projects. Also note, that collision detection is mostly implemented together with physical simulation, even though both can be used separately without one another they mostly are used together. In case of Panda3D's physic ans collision implementation this is not the case as extreme as with bullet and ode, as you need to enable physics and collision detection separately. Now, to add collision detection we need to do a few things. Dependent on which collision system we take there are other methods to handle the collisions and implementation. For this tutorial Panda3D's own implementation on collision detection will meet all our needs as well as being rather simple to use and hence we will use it to enable collision detections.

The first thing you need to do with each collision and physics system is to enable the system we want to use. For Bullet and ODE this would mean we need to instantiate a PhysicsWorld which would eiter be of type BulletWorld or OdeWorld. These would then handle all physic simulations and collision detection math that will happen in this world instance. Most applications will only need one physic world instance, but if you really have the need you can instantiate as many as you like. For Panda3D's implementation of collision detection as it isn't tied to physic simulation as in the other two libraries, we don't need a specialized world instance. For it we just need to initialize so called collision traverser. Most of the time a single traverser will be enough for the needs of a game but as with the physics world you can create as many as you think you need. Due to it's tight integration, there already are some convenient things for you. First you don't need to add a new task that will update the physic world of your game if you set the traverser in the global variable base.cTrav; the traverser set in this variable will automatically be updated every frame. You just need to keep track on traverser you created additionally to that one.

To get that now in our tutorial code we have to add a few things.

First add a new import statement in the main.py script so we have the CollisionTraverser class available as this is defined in the panda3d.core, we can simply extend that statement as follow.

```
from panda3d.core import (
    CollisionTraverser,
    CollisionHandlerPusher,
```

Next, we need to put these lines in the postInit method of the Main class somewhere before the

player initialization and generally before you actually add any collision solids. Best put it at the beginning of the # initialize game content section.

```
base.cTrav = CollisionTraverser("base collision traverser")
base.pusher = CollisionHandlerPusher()
```

In the first line we create a Traverser that will as described above handle the collision math and will check all “colliders” or also called “from objects” given to it. Later on, you can decide which objects get added to it and hence become from objects. All other objects, not added to it will become so called “into objects”. Mostly from objects are movable objects and into objects static ones but that's not a necessity.

In the next line we instantiate a collision handler or to be more specific a pusher collision handler which is one out of five specialized handlers. The others are CollisionHandlerQueue, CollisionHandlerEvent, PhysicsCollisionHandler and CollisionHandlerFloor. Following is a short description of each of the collision handlers.

The queue handler is the most simplistic one. It just records all collisions that happen to the from object set to that handler.

The event handler will send events dependent on what event wildcards where set to the handler. For example if we have a from collision object for a player called “ColPlayer” and an into collision object for a switch within a level called “ColSwitch1”. Then we could add an event wildcard like “%fn-into-%in” with the handlers .addInPattern method. For the example this would result in an event named “ColPlayer-into-ColSwitch1” which get thrown whenever the player moves into the switch collision object. For mor information about what can be used to create the wildcard text or which add*Pattern methods are available you can consult the Panda3D manual section named “Collision Handlers”.

The pusher handler acts like the event handler. In addition to that the pusher handler has the specialty to keep the objects given to it apart so that they won't slip through other collision solids. This is for example really useful for characters as they should not be able to move through walls, other characters or any other solid object. Hence we will also use that for the characters in this tutorial.

The physics handler is the most specialized handler. It acts just like the pusher handler but keeps track on Panda3D's physics system. It also only accepts physically movable objects (objects of type ActorNode, not to confuse with the Actor class which is for loading and playing models and animations) as it's from objects.

The floor handler is also a very specialized handler and is dedicated to check as the name states if a from object collides with the level's floor. Therefor you would normally give the handler a ray or infinitely long line pointing down towards the floor from the object you want to be placed on the ground. The handler will then automatically place that object instantly or if configured correctly move the object slowly towards the floor.

Having the collision system prepared, we now can add some solids to it to actually check collisions with it.

The first problem we currently have, that we'll tackle will be the ability to walk through the other character. This will be rather simple as we have set up a pusher collision handler which will do most of what we want automatically and we just need collision solids for the characters.

Move into the player.py script and add the following things.

As in the other sections, we first need to import the new classes we'll be using for collision detections. So extend the panda3d.core import line as follow

```
from panda3d.core import (
    CollisionSphere,
    CollisionNode,
    KeyboardButton)
```

Nextt, we add the collision solid to the character. For now we will use only one sphere solid. The sphere solid is the most efficient collision solid of them all and you should use it whenever possible. We will put the sphere in the upper bodypart of the character floating over the ground and reparent it to our moving actor so the sphere always is at the same position as our visible character model.

Add these lines at the bottom of the Player class's init method or somewhere below the actor setup

```
characterSphere = CollisionSphere(0, 0, 1.0, 0.5)
self.collisionNodeName = "character%dCollision"%charId
characterColNode = CollisionNode(self.collisionNodeName)
characterColNode.addSolid(characterSphere)
self.characterCollision = self.character.attachNewNode(characterColNode)
base.pusher.addCollider(self.characterCollision, self.character)
base.cTrav.addCollider(self.characterCollision, base.pusher)
```

Let's go through this lines and see what they do.

In the first two lines, we prepare the variables that we'll need to add a collision solid to our character. First a collision sphere as mentioned above which has a radius of 0.5 units and floats 1 unit above the ground. After that we set a name for our collision node which we will also need later for the attack hit handling. The Collision node is a specialized Panda Node that holds all our collision solids and hence has some dedicated functions for that, which brings us to the next line.

Having a collision solid and -node we need to add the solid into that node. This is achieved by calling the nodes addSolid method with the just created collision sphere.

Following to that we attach the collision node as child to our actor node, self.character. This way the collision node will always be placed and moved respective to the character as well as hinder the character from moving through other solid objects.

Then we need to add the nodepath we just got from the attaching the collisionnode to the actor as well as the actor itself to the pusher, so that the sphere and actor will always pushed out of other

objects.

Finally add the nodepath to the traverser and the handler you want to get responsive to that object so the nodepath will also be handled as into node and other characters and objects can bump into it.

As a side note, you can also add this method after the collision node has been instantiated to get a visual representation of your collision solid.

```
self.characterCollision.show()
```

But be aware that this will, if there are many collision solids, decrease your performance quite a bit.

Getting so far, you should now have two characters that cannot move through one another if you restart your application.

But that's not all we want to achieve with the collision system. As our little game should be a beat em up game, we also use our collision detection system for hit detection.

There are many ways to achieve this and it all depends on how accurate you want to have your hit detection. As our game so far is rather simple, our hit detection will also be really simple. We will use a ray segment with a given length that will point forward from the character towards the other character. If that ray hits the other character and an attack action is made, we will hit the other character. Indeed you could also add a more complex system where you attach smaller collision solids to hand, feet and other body parts bones of the character which will greatly enhance the accuracy of the detection, but we'll leave that up to you if you want to implement this amount of hitboxes. After this section, you should at least know enough to being able to add those.

To add the mentioned ray, you need to add these lines of code to the end of the Player class init method.

```
characterHitRay = CollisionSegment(0, -0.5, 1.0, 0, -0.8, 1.0)
characterColNode.addSolid(characterHitRay)
```

Here we create a collision segment which gives us a line starting at the first three parameters (x,y,z) and stretches to the last three parameters (also x,y,z). This ray then got added to our characters collision node where we already added the characters sphere to.

Now we need a few more functions to handle the attack behavior and let the character know when he gets attacked:

```
    def setEnemy(self, enemyColName):
        self.enemyColName = enemyColName
        inEvent = "%s-into-%s"%(enemyColName,self.collisionNodeName)
        base.pusher.addInPattern(inEvent)
        self.accept(inEvent, self.setCanBeHit, [True])
        outEvent = "%s-out-%s"%(enemyColName,self.collisionNodeName)
        base.pusher.addOutPattern(outEvent)
        self.accept(outEvent, self.setCanBeHit, [False])
```

```
    def setCanBeHit(self, yes, collission):
        eventName = "hitEnemy%s"%self.collisionNodeName
        if yes:
            self.accept(eventName, self.gotHit)
        else:
            self.ignore(eventName)
        self.canBeHit = yes

    def gotHit(self):
        if not self.canBeHit or self.isDefending: return
        self.request("Hit")
```

Here we make great use of Panda3Ds event system. In the first function, setEnemy we add events to the pusher collision handler that will be thrown whenever an enemy character collides with this characters collision node. Directly in the lines after we set up the “in” and “out” patterns straight use them to call our setCanBeHit function with the respective parameters passed to it. This setCanBeHit function is the next in line. It checks if the player can be hit and respectively activates or deactivate the catching of the hitEnemy* event as well as set a variable to double check if this player can be hit.

To make sure the evens won't be caught at a later time after the character has been removed, add the following line in the characters stop method.

```
self.ignoreAll()
```

This will make sure no events will be caught by the character anymore from there on.

Finally we come to the function where the player actually got attacked. In the first line we check if the player can be hit or blocks the attack. In the next line we request the hit state of the character as he got hit.

Having the functions, we now also need to call them at the right places in our application. Let's start with the calls in the Player class itself. In the start method somewhere before we add the moveTask task to the task manager, add this one line

```
self.canBeHit = False
self.isDefending = False
```

Here we have two variables which we use to determine if the character can be hit by the enemy character.

Within the moveTask method

```
        ...
        if isDown(self.defendButton):
            if self.state != "Defend":
                self.isDefending = True
```

```
                self.request("Defend")
            return task.cont
        self.isDefending = False
        ...
        if isAction:
            base.messenger.send("hitEnemy%s"%self.enemyColName)
            return task.cont
```

in the first part of the addition, we set the isDefending variable to True and let it be set as long as the player is in the defense state.

In the next part, we send the hitEnemy event whenever the player presses one of the attack keys. It doesn't matter for this player if or who catch the event so we just send it.

Finally we need to let the characters know about their enemy by calling the setEnemy method of the Player class from the Main class by adding those two lines right after both players have been set up.

```
self.player.setEnemy(self.player2.collisionNodeName)
self.player2.setEnemy(self.player.collisionNodeName)
```

These two lines make sure the characters know about each other and can send the respective events as well as know when they collide and hence be able to hit each other.

Now that we have that, we need some stats like for the players health to determine if the player got defeated.

Let's add the players health as a variable which will represent the health in percent from 0-100 and change it whenever the player got hit.

First simply add this to the start method of the Player class somewhere before we add the moveTask task to the task manager,

```
self.health = 100
self.gotDefeated = False
```

and change the gotHit function to

```
    def gotHit(self):
        if not self.canBeHit or self.isDefending: return
        self.health -= 10
        if self.health <= 0:
            self.gotDefeated = True
            self.request("Defeated")
        else:
            self.request("Hit")
```

With our newly added health variable, we are now able to change the players health whenever he got hit. If the player got hit he always loose 10% of it's health until it reaches 0. When the players health reaches 0 we directly move to the defeated state and also set the respective variable for being

defeated to True.

Finally again in the moveTask function we will check for the gotDefeated variable by adding this directly at the top of the function

```
if self.gotDefeated:
    base.messenger.send("GameOver")
    return task.done
```

Here we just check if we got defeated and if so, send a GameOver event which we will use later as well as stop the moveTask task so the player cannot move anymore and not get into another state.

Now that we have all that, the player controls are done, you can move and fight with your characters as well as defeat each other with 10 hits. If a player got defeated you still need to restart the application but we'll take care about that later. As you may noticed, if you move out of the screen, the camera doesn't get updated which you might await from such a game, this will be our next part.

Camera control

In this section we will talk about another part of games that has to be done to get a good gameplay, the camera control. Having a good camera control and placement, no matter if controlled by the user or only by the application, is, as each part of a game, a really important thing. For each game, the camera must be smooth and not distracting to the players gameplay as otherwise the game may be hard to play and just look and feels unprofessional. Imagine a 3D platformer where the camera is always at a fixed position in the world and just look at the player no matter how far he is away. It could get really hard to time the jumps from and to a platform if the player even stays in view all the time, or imagine a game with a really shaky camera or floating through just any object so the player can see the things behind it or rather in it.

For our game we create in this Tutorial, we will have a quite simple camera setup. It will ensure the players are always visible and tries to always stay at a reasonable distance to them. Due to this simple setup of camera behavior and also due to the fact that our camera is not bound to just one character, we can't place it within a player related script but have it in a more global place and hence will put it in the Main class. If your camera setup gets more complex and exceed the amount of functionality we put into this Tutorial you should make sure to put your camera code into a dedicated script. But for now, to reach the things we want, just add the following function somewhere in the # BASIC FUNCTIONS section of the Main class.

```
    def updateWorldCam(self, task):
        playerVec = self.player.getPos() - self.player2.getPos()
        playerDist = playerVec.length()
        x = self.player.getX() + playerDist / 2.0
        self.camera.setX(x)
```

```
        zoomout = False
        if not self.cam.node().isInView(self.player.getPos(self.cam)):
            camPosUpdate = -2 * globalClock.getDt()
            self.camera.setY(self.camera, camPosUpdate)
            zoomout = True
        if not self.cam.node().isInView(self.player2.getPos(self.cam)):
            camPosUpdate = -2 * globalClock.getDt()
            self.camera.setY(self.camera, camPosUpdate)
            zoomout = True
        if not zoomout:
            if self.camera.getY() < 0:
                camPosUpdate = 2 * globalClock.getDt()
                self.camera.setY(self.camera, camPosUpdate)
        return task.cont
```

This task function will handle all our camera handling for the game. It will move the camera back when one or both players left the viewable area and moves closer as long as they are inside the visible area. Also it will keep the camera in the middle of both characters.

But let's get through this one line after another.

```
playerVec = self.player.getPos() - self.player2.getPos()
playerDist = playerVec.length()
x = self.player.getX() + playerDist / 2.0
self.camera.setX(x)
```

These few lines let the camera be positioned in the center calculated by both the positions of the player characters. The first line calculates a 3D vector by simply subtracting one position from the other one. The next line gives us the length of the vector, hence the distance the characters are away from each other. Then we calculate the x position. Here we assume that the first player is always on the left side of the screen and hence we can just add half the distance of the player to that one to get the center x position. Finally we simply set the cameras x position to the newly calculated one.

```
if not self.cam.node().isInView(self.player.getPos(self.cam)):
    camPosUpdate = -2 * globalClock.getDt()
    self.camera.setY(self.camera, camPosUpdate)
    zoomout = True
```

Next we have two blocks, checking if the character is inside the viewable area of the camera and otherwise move the camera back a bit. The check is made quite simple by calling a function of the camera class. To get this class we have to first call the node() method on the globally stored cam NodePath. The isInView method takes a position and returns a boolean value which determines if that position is visible by the camera or not. Also note, that we need to get the players position relative to the camera, which is done by passing the cam node to the getPos method, as it won't work correctly otherwise. If we have done that check and the character really is outside of the viewable area, we calculate an amount of which the camera gets moved. This is done the same way as we already did inside the Player class's moveTask, by getting the DT value and multiply it with a defined movement speed. Also same as in the Player class, we apply the new value to the camera,

respective to itself so it moves backward. Finally we set the zoomout value to True as we now moved the camera back a bit. This value will be checked later on in the following check.

```
if not zoomout:
    if self.camera.getY() < 0:
        camPosUpdate = 2 * globalClock.getDt()
        self.camera.setY(self.camera, camPosUpdate)
```

This last few lines let the camera move closer to the characters if the camera doesn't move back and hence the characters are within the visible bounds of the camera as well as is further away then the set minimum position of 0. The position calculation and placement of the camera is done the same way as before just in the positive direction.

After that we add the usual return task.cont as we want the task to run until it got manually stopped.

You may notice, that we have called getX and getPos directly on the Player class, even tho they not exist directly in the Player class. To make them available in the player class we need to forward the correct functions within it to make it possible to control the character more simple. These two functions as well as any others can easily be forwarded by adding the following two lines in the __init__ method of the Player class.

```
self.getPos = self.character.getPos
self.getX = self.character.getX
```

Finally we just need to add the task to the task manager by simply putting the following line at the end of the enterGame method.

```
self.taskMgr.add(self.updateWorldCam, "world camera update task")
```

and respective to it this line at the start of the exitGame method.

```
self.taskMgr.remove("world camera update task")
```

That's all for our simple camera setup and it will be good enough to not get into the way of the players. Also that's a rather common way to handle camera movements in game genres like the one this game falls into. There sure could be done some refinements or enhancements, but those will then fall into the dedicated section of this project.

Having all that, we have two fully working characters and can already play the game with its main gameplay. But it just don't feel complete yet and even if we have already got a bunch of things done, there's much more to add to an interesting and full featured game. The next section will bring a bit more color to our game by adding a nice background to it so it doesn't look like our characters are floating and fighting in a pitch black space scene like the end of time has come.

The environment

Having the main gameplay of our game we move on to the next important part. Creating environments for it.

Level design is a really big and important part of game creation for many genres. They bring in much of the look and feel for your game and the better the environments are, the deeper the player gets drawn into the game.

Creating the environment

Same as in the character creation section, we won't get into that much detail about level creation within this tutorial but instead give you a few hints for level creation and special things you should keep in mind when building levels for Panda3D or generally games.

Most of the hints stated in the character creation section also applies by creating levels but here are a few more specific to levels.

1. Separate as many assets as possible and link them into your level. In blender using yabee, make sure to add a game property of type string named “file” and set it to the path of that asset's egg file best without the egg extension. For example like “file” - “levels/level_base_assets/ladder” where ladder will be an existing model file called either ladder.egg or ladder.bam or any of the other model files Panda3D recognizes as such.
2. You should have visible objects and collision solids separated in separate layers as well as base level meshes and other other objects in it.
3. Places empty nodes at important positions if necessary like the players start position or positions where collectibles should be put at.
4. If you place collision objects and you're using blender with yabee, make sure to add the correct game properties so that they will be exported correctly. For example you can add a game property of type string called ObjectType with level as it's value and the selected object will be an invisible collision solid of type level, which means all faces will point upward so a character could walk and stand on slopes (note, this way a character could even move up walls sometimes). You can also add a property named Collide and add the existing egg collision values into it. More about the parameters you can set as well as other ObjectTypes can be found within the Panda3D Manual here: www.panda3d.org/manual/index.php/Egg_Syntax as well as in the Confauto.prc file which gets installed with the Panda3D sdk.
5. Transparent Objects should have a set of game properties assigned to them in order to be correctly drawn. Those are in “property name – value” order, “blend – ADD” ; “blendop-a –

INCOMING-ALPHA" ; "blendop-b – ONE" ; "bin – fixed". Note that you may need to change these if you encounter alpha rendering problems in the game to fit your needs.

Loading the environment into the game

Having some level models now, we need to get them into our game, so take them and put them into the correct place.

For the Levels, we will add a few new directories so everything has its structure. Inside your assets folder, create the following directory structure: levels/arena1/ and put all your arena1 file in there. Do the same for all other arenas you have made and name the folders respectively arena2/ arena3/ etc.

Now let's bring this into our application. We will do this by adding a new script called arena.py in the source root folder …/game/ which will then handle the loading of our arena models.

Open that new script and add the following base to it:

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

class Arena:
    def __init__(self, arenaNr):
        arenaPath = "levels/arena%d/" % arenaNr
        self.arena = loader.loadModel(arenaPath + "arena")
        self.arena.setScale(2)
        self.arena.reparentTo(render)
        self.arena.hide()

    def start(self):
        self.arena.show()

    def stop(self):
        self.arena.hide()
```

Most of this code shouldn't be new to you if you followed this tutorial until now. The only new lines are the one where we load the model using loader and the setScale. The loader is another global variable which we got through instancing the Panda3D engine it is used to load static model files. The setScale method is used to change the models size with the given factor. Here we set the scale to two so the model is twice as big as it actually is defined in the model file.

Next we will add a new method to the class that we will use to place the players in the arena. This function should look as follow.

```
    def getStartPos(self, charNr):
        if charNr == 1:
```

```
            return self.arena.find("**/StartPosA").getPos() * 2
        elif charNr == 2:
            return self.arena.find("**/StartPosB").getPos() * 2
        else:
            return (0,0,0)
```

This code will return the start positions set in the model file of the arena. Note that we have to apply the scale as set in the init method to the arena as it isn't applied to the getPos method.

Using the find method we can search through a NodePath for a specific named node in our case this is StartPosA and StartPosB which both are empty nodes in our arena model. Using **/ in the search string will let the find method search upward through all branches of the given NodePath until it found one match. Note that this method will only return the first matching node. Note, there are other functions which are able to return a list of nodes matching but we don't need them yet.

To be a bit more error resistant, if we pass a wrong character number to the method, we return the center as the starting position.

Now let's use this in our main class to load the arena and position the characters within it. Open your main.py script and add this line somewhere around your already existing Player import statement.

```
from arena import Arena
```

Next we need to instantiate the Arena class and use it, which we will all do within the enterGame method, so change it as follow:

```
        self.arena = Arena(1)
        self.arena.start()
        self.camera.setPos(0, -5, 1.25)
        self.player.start(self.arena.getStartPos(1))
        self.player2.start(self.arena.getStartPos(2))
```

This will load the arena and render it as well as place the characters within it and the camera a little bit back and up so the arena as well as the characters are clearly visible.

Now we also need to “fix” our camera behavior a little as it otherwise will try and move much to close to the players. So let's move into the updateWorldCam task function and change the line where we check the cameras Y position so that it is always at least 5 units (as seen from the global coordinates -5 units) away from the center and hence from our characters:

```
        if not zoomout:
            if self.camera.getY() < -5:
                camPosUpdate = 2 * globalClock.getDt()
```

That should be all for now to get our arena into the game as well as having everything working again, so lets try and restart the application.

It does look better, doesn't it, still there's missing something, so lets take a look at making the environment look a bit more realistic by adding light to it.

There are a bunch of lamps you can add to your game to add light to your objects but keep in mind, many lights will drop your applications performance so make sure to plan how and where you place your light. On another note, only a few lamps will produce drop shadows and this also not by default but more of that later.

First we need to import the lamp types we want to use, which would be ambient and spotlight. To see what other lamps are available, check this part of the Panda3D manual http://www.panda3d.org/manual/index.php/Lighting. Now add the following lines in your Arena.py script to make the desired lamps available in it.

```
from panda3d.core import (
    AmbientLight,
    PerspectiveLens,
    DirectionalLight)
```

Having them, we will add the lamps to our arena.

First we add the ambient light which we simply can do with the following lines, simply put them below the arena loading in the init method.

```
ambientLight = AmbientLight('ambient_light')
ambientLight.setColor((0.2, 0.2, 0.2, 1))
self.alnp = render.attachNewNode(ambientLight)
```

We instantiate the ambient light with a name, set it's color and finally attach it to our scene.

Next we going to add the spotlight which will represent the sun.

```
sun = DirectionalLight("sun")
sun.setColor((1, 1, 1, 1))
sun.setScene(render)
self.sunNp = render.attachNewNode(sun)
self.sunNp.setPos(-10, -10, 30)
self.sunNp.lookAt(0,0,0)
```

After instantiating the spotlight, we set the color of it to white (parameters are r,g,b,a). Next we need to instantiate a lens which is a specialty of the spotlight as it uses a lens to set the cone of light the lamp will produce. Following we attach the lamp node to render. Then we just set the lamps position and HPR values which we do by using the lookAt method that let the objects +Y axis point toward the given point.

With those lines we added lamps to the game, though, they won't have any effect yet, we need to set those lights on the nodes that should get lit from them. So add the following lines to the start function.

```
render.setLight(self.alnp)
render.setLight(self.sunNp)
```

With this line we set the light to light up the render node and hence all its child nodes, so every object which is reparented to render will be affected by the light. Note, if you have light that generates a shadow, you should search for the closest objects around it and set the light on those.

As we also need to remove those lights by ourself if we stop the arena, we need to add the following line to the stop function

```
render.clearLight()
```

This will remove all lights set on the render node and as we only set lights here in the arena script, we can securely call this. Also we need to do this if we later want to start and stop the level more than once, otherwise lights will be added whenever the level is started and they all get added to render.

Now if you restart the application, it should look much better with all the new shadows the lights produce.

Further shading will be handled at a later section of this tutorial. So lets continue with the next part.

Building the GUI

As you may know, most games don't start directly into the game. They all have at least a main menu which they show up at startup. If you look at bigger games, they also all tend to have a title screen, option menus, help and loading screens and much more. This tutorial will show you how you can generally set up such UIs so you can add as many of them as you like.

Panda3D comes with two GUI toolkits, one is Rocket, which is a GUI toolkit that uses html and css like syntax to describe the interface layout and the other is Panda3Ds internal system called DirectGUI. The DirectGUI layout can be described with python code.

For this tutorial we will use Panda3Ds internal GUI as it is more easy to use for small and simple UIs and also is better integrated as the Rocket GUI.

Lets start with adding a very basic main menu to our application we made so far.

Adding a simple main menu

Every game should at least have a main menu to re-/start a game and quit the application. Later on you can also extend it with useful information for the users like version number, a clock with the actual time, an option menu and much more. For now we will add a menu that shows up directly after starting the game and displays two buttons, one for starting the game and one for closing the application. We will also let the application return to the menu when the game is over so the player can start a new game without restarting the whole application.

We will put the menu in a new script file, so create a menu.py within the source root folder, …/game/. The following code will create the class for the menu we will be using, put it in the file you just created.

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

from direct.gui.DirectGui import (
    DirectFrame,
    DirectLabel,
    DirectButton)

class Menu:
    def __init__(self):

        self.frameMain = DirectFrame(
            frameSize = (base.a2dLeft, base.a2dRight,
```

```
                        base.a2dTop, base.a2dBottom),
            frameColor = (0, 0, 0, 0))
        self.frameMain.setTransparency(1)

        self.title = DirectLabel(
            scale = 0.25,
            pos = (0.0, 0.0, base.a2dTop - 0.25),
            frameColor = (0, 0, 0, 0),
            text = "The Game",
            text_fg = (1,1,1,1))
        self.title.setTransparency(1)
        self.title.reparentTo(self.frameMain)

        self.btnStart = self.createButton(
            "Start",
            .25,
            ["Menu-Start"])

        self.btnExit = self.createButton(
           "Quit",
            -.25,
            ["Menu-Quit"])

        self.hide()

    def createButton(self, text, verticalPos, eventArgs):
        btn = DirectButton(
            text = text,
            scale = 0.25,
            pos = (0, 0, verticalPos),
            command = base.messenger.send,
            extraArgs = eventArgs,
            rolloverSound = None,
            clickSound = None)
        btn.reparentTo(self.frameMain)

    def show(self):
        self.frameMain.show()

    def hide(self):
        self.frameMain.hide()
```

Lets go through this and see what's happening in there.

The first few lines is some basic python scripting. We just import some classes that we need to set up our menu and define a class.

Within the init method you see how we create the menu elements. At first we create a frame.

```
        self.frameMain = DirectFrame(
            frameSize = (base.a2dLeft, base.a2dRight,
                        base.a2dTop, base.a2dBottom),
            frameColor = (0, 0, 0, 0))
```

A frame is like a rectangular container for other GUI elements it is also used as the base of most of

the other DirectGUI elements and hence they all have functions available to the DirectFrame. As parameters for our main frame, we don't need much yet, we just set a frame size and the color of it to transparent black. The size of the frame is set by a four tuple parameter as you may see, ordered left, right, top and bottom using pandas units for the size. A total square window would have -1, 1, 1 and -1 as the maximum number of visible units in it tho these change dependent on the size of the screen. For this we use some variables set in the global base member, there are a few a2d* variables for all sides and edges of the window like a2dLeft a2dTopRight and so on. Note, the variables get reset whenever the applications window is resized but the frame itself won't change it's size automatically. You can take a look at the aspectRatioHandling.py script in the panda3d-code-collection on Launchpad.net to get an idea of how you can handle window resizes.

```
        self.title.setTransparency(1)
```

As we set the color of the frame to transparent, we also need to activate the transparency attribute on the GUI element as otherwise it'll be opaque all the time.

Now you might have noticed, that we doesn't reparent the Frame to anything throughout the class. This is due to the fact, that all DirectGUI elements get automatically reparented to aspect2d which is another globally available base nodepath like render is, just with the difference, that this nodepath is for rendering 2D elements and respects the aspect ratio of the screen/window, hence also the necessity to use the base.a2d* members in the frame sizing.

```
        self.title = DirectLabel(
            scale = 0.25,
            pos = (0.0, 0.0, base.a2dTop - 0.25),
            frameColor = (0, 0, 0, 0),
            text = "The Game",
            text_fg = (1,1,1,1))
```

Next we add a label to our menu. A label represents a simple text that gets displayed on the screen. Some of the variables set in a DirectLabel or general GUI element are just like the members you can also set in a NodePath like positioning, scale and rotations. As a directLabel by default is rather big in size, we decrease its scale quite a bit and then position it a bit lower than at the top edge of the screen. Note that the horizontal position is set via the x-axis and the vertical position via the z-axis. Also just like the frame we had before we set the labels background color to transparent black. Here you also note that DirectLabel is based on DirectFrame, all frame members can be accessed by a preceding frame* like frameColor, frameScale etc. Like the frame specific parameters, you can also access the text specific parameters by a preceding text_* whereas text_fg represents the text color (fg = foreground). We use that to set the text color to a fully opaque white and the text itself should read “The Game”.

```
        self.title.reparentTo(self.frameMain)
```

Now as we don't want to have the title reparented directly to aspect2d but instead have our frame as

parent, we reparent the title to it. This has one major advantage, if we want to show or hide the menu, we just need to show or hide the main frame and all added GUI elements will be shown or hidden respectively.

Now we get to the Buttons of our menu. As we want to add multiple buttons to the menu we write a small helper function that will create a button for us with a few given parameters.

```
    def createButton(self, text, verticalPos, eventArgs):
        btn = DirectButton(
            text = text,
            scale = 0.25,
            pos = (0, 0, verticalPos),
            command = base.messenger.send,
            extraArgs = eventArgs,
            rolloverSound = None,
            clickSound = None)
        btn.reparentTo(self.frameMain)
```

Here we create a DirectButton which resemble a clickable frame that has a few graphical additions to have the look and feel of a button. The text is set as a centered black label directly on the button, as with the label, the default size of a button is pretty big, hence we scale it down a bit. Then as we want all buttons be positioned vertically on the center of the screen, we just let the z-axis be changeable. After that, the next two variables belong together, the command is a function that get called whenever the button got clicked and the extraArgs is a list of anything you want to pass to the function called in the command. Here we also have a first glance at the event system of Panda3D with the function base.messenger.send, you can send an event that you can later catch wherever you see fit. For this events, we will catch them in a bit within the main class. The last two arguments we set are sounds that get played if the user moves the mouse over the button and the other plays when the player clicks the button. As we don't want the default Panda3D plop sound we set them to None.

Now just like the label, we add the button to our main frame which get usefull in the two following functions:

```
    def show(self):
        self.frameMain.show()

    def hide(self):
        self.frameMain.hide()
```

Now you can see how simple it is to show and hide the complete menu by just calling the show and hide function on our main frame.

Now this menu is fully functional, we just need to add it into our game by changing a few things in our main class.

First of all, we add a new state to our main state machine and name that state Menu. Just add these

two functions at the start of the # FSM PART

```
    def enterMenu(self):
        self.accept("Menu-Start", self.request, ["Game"])
        self.accept("Menu-Quit", self.quit)
        self.menu.show()

    def exitMenu(self):
        self.ignore("Menu-Start")
        self.ignore("Menu-Quit")
        self.menu.hide()
```

If we enter the Menu state we will keep track on the events we send from within the Menu class via the base.messenger.send function by using the self.accept method. The self.accept method is a function that gets available through the DirectObject class which is one of the base classes of Panda3D and hence many objects have the accept function. The accept function awaits the name of the event that get raised, a function that should be called whenever the event got caught and an optional list of extra parameters that should be passed to the function. In the end of the enterMenu function we simply show our Menu.

In the exitMenu method we ignore the events again, so if events with the same name get thrown after that, the functions as given in the earlier self.accept methods won't be called. Same as the accept method the ignore method is a function from the DirectObject class. After all that, we hide the menu again.

Now that we have a new state, and as we want the menu to be the first thing the user sees after starting the game, we change the first request in our application, the one at the end of the Main class's postInit method to the following.

```
        #
        # Start with the menu
        #
        self.request("Menu")
```

Now the menu should show up if you restart your application and if you hit start you should get to the main gameplay whereas quit will stop the game. Now if you hit escape within the main game, the application quit and if you hit escape within the menu, you get directly into the game, which is not quite the behavior one would expect, so let's change this. Within the Main class, we have a function called __escape, which is always called when the escape key on your keyboard got hit. If you change the “Game” to “Menu” as seen in the following lines of code, the behavior should be much more pleasant to the player as it is something we know from other games. While in the menu the application will quit and if we are not in the menu state, we will move to it.

```
    def __escape(self):
        if self.state == "Menu":
            self.quit()
        else:
```

```
            self.request("Menu")
```

That's all for our small menu, now we will add a few other UIs that a player awaits if he plays a game like ours.

Finally let's add a bit to the game state exit function, so the level as well as the characters get cleaned up correctly and we start with a fresh game whenever we return and restart from the menu.

So replace the pass in the exitGame function with these lines

```
self.player.stop()
self.player2.stop()
self.arena.stop()
```

From here on the menu should be fully functional and working. Now we can take a look at making it's design more pleasant to the eye by adding a nice background and some nice button style. For this new style, just change the code of the menu file to the following:

```
    def __init__(self):

        self.frameMain = DirectFrame(
            image = "assets/gui/MenuBackground.png",
            image_scale = (1.7778, 1, 1),
            frameSize = (base.a2dLeft, base.a2dRight,
                         base.a2dTop, base.a2dBottom),
            frameColor = (0, 0, 0, 0))
        self.frameMain.setTransparency(1)

        self.title = DirectLabel(
            scale = 0.15,
            text_align = TextNode.ALeft,
            pos = (base.a2dLeft + 0.2, 0, 0),
            frameColor = (0, 0, 0, 0),
            text = "Main Menu",
            text_fg = (1,1,1,1))
        self.title.setTransparency(1)
        self.title.reparentTo(self.frameMain)

        self.btnStart = self.createButton(
            "Start",
            -.10,
            ["Menu-Start"])

        self.btnExit = self.createButton(
           "Quit",
            -.25,
            ["Menu-Quit"])

        self.hide()

    def createButton(self, text, verticalPos, eventArgs):
        maps = loader.loadModel("assets/gui/button_map")
        btnGeom = (maps.find("**/btn_ready"),
```

```
                maps.find("**/btn_click"),
                maps.find("**/btn_rollover"),
                maps.find("**/btn_disabled"))
        btn = DirectButton(
            text = text,
            text_fg = (0,0,0,1),
            text_scale = 0.05,
            text_pos = (0.02, -0.015),
            text_align = TextNode.ALeft,
            scale = 2,
            pos = (base.a2dLeft + 0.2, 0, verticalPos),
            geom = btnGeom,
            relief = 0,
            frameColor = (0,0,0,0),
            command = base.messenger.send,
            extraArgs = eventArgs,
            pressEffect = False,
            rolloverSound = None,
            clickSound = None)
        btn.reparentTo(self.frameMain)
```

We set the image of the background directly to our main frame, each directFrame has aside of a text also an image it can hold which will be displayed within that frame. We set the images x scale to 1.7778 due to the fact that our image is 1920x1080px in size and hence results in a aspect ratio of 1.7 periodic to 1 or the more common 16:9. On screens with differing aspect ratio like the also still very common 4:3, the image will be fitted into the frame. So for example on a 1024x768 you will see the background color or things behind the image at the top and bottom side. If you have set the background color of your application to black, this will give your main menu two black bars which you might know from watching movies. If you want to have another behaviour, you need to implement that by yourself, or use a 3D scene as background which will then be seen through the 3D camera and can be fitted to any screen sizes.

The buttons and the title will be moved to the left and shifted a bit down. Also we change the titles text as it is already given in the background image. The buttons get a nice texture, which we load through a specialized egg file which holds a few frames each with another button state. These egg files are made with the egg-texture-cards command that comes with the Panda3D SDK. To know how this egg file was created exactly, take a look at the create buttonmap.txt file located in the Design directory that comes with the source of this Tutorial. Using that egg file, we split the separate button states into a tuple and hand it over to the buttons geom parameter. With that we just need to change the frames color as we want the buttons to apear transparent and change the text style to fit the buttons. Also we disable the pressEffect which would otherwise scale the button a bit down when the user clicks it to give the appearance of “pressing” the button inside the screen.

The image and egg file used for the menu are stored in the new assets/gui/ directory. This will be used for any of our GUI artwork.

Character selection

As we have multiple characters in our game and the players should be able to choose from them, we will add a small menu where the player can choose which one to play. Therefor we create a new script for the selection screen and add another state to the Main class's FSM just like we did before with the main Menu.

Our menu will consist of a three column layout, whereas the left and right column will show the players selection where the middle one will be used to show all available characters.

The following code will set up a completely styled character selection menu for us so simply put that into a new file called characterselection.py in the same directory as you already put the menu.py.

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

from panda3d.core import (
    TextNode,
    Texture)
from direct.gui.DirectGui import (
    DirectFrame,
    DirectButton,
    DGG)

class CharacterSelection:
    def __init__(self):

        self.frameMain = DirectFrame(
            frameSize = (base.a2dLeft, base.a2dRight,
                         base.a2dTop, base.a2dBottom),
            frameColor = (0.05, 0.05, 0.05, 1))
        self.frameMain.setTransparency(1)

        width = abs(base.a2dLeft) + base.a2dRight

        red = loader.loadTexture("assets/gui/CharRedBG.png")
        red.setWrapU(Texture.WM_repeat)
        red.setWrapV(Texture.WM_repeat)
        self.char1Frame = DirectFrame(
            text = "Player 1",
            text_fg = (1,1,1,1),
            text_scale = 0.1,
            text_pos = (0, base.a2dTop - 0.2),
            frameSize = (-width/6.0, width/6.0,
                         base.a2dTop, base.a2dBottom),
            frameTexture = red,
            pos = (base.a2dLeft+width/6.0, 0, 0))
        self.char1Frame.updateFrameStyle()
```

```
        self.char1Frame.setTransparency(1)
        self.char1Frame.reparentTo(self.frameMain)

        blue = loader.loadTexture("assets/gui/CharBlueBG.png")
        blue.setWrapU(Texture.WM_repeat)
        blue.setWrapV(Texture.WM_repeat)
        self.char2Frame = DirectFrame(
            text = "Player 2",
            text_fg = (1,1,1,1),
            text_scale = 0.1,
            text_pos = (0, base.a2dTop - 0.2),
            frameSize = (-width/6.0, width/6.0,
                         base.a2dTop, base.a2dBottom),
            frameTexture = blue,
            pos = (base.a2dRight-width/6.0, 0, 0))
        self.char2Frame.setTransparency(1)
        self.char2Frame.reparentTo(self.frameMain)

        self.footerFrame = DirectFrame(
            text = "PLAYER 1 - CHOOSE YOUR CHARACTER",
            text_fg = (1,1,1,1),
            text_scale = 0.08,
            text_pos = (0, -0.03),
            frameSize = (base.a2dLeft, base.a2dRight,
                         -0.1, 0.1),
            pos = (0, 0, base.a2dBottom + 0.2),
            frameColor = (0, 0, 0, 0.5))
        self.footerFrame.setTransparency(1)
        self.footerFrame.reparentTo(self.frameMain)

        self.charSelectFrame = DirectFrame(
            text = "VS",
            text_fg = (1,1,1,1),
            text_scale = 0.1,
            text_pos = (0, base.a2dTop - 0.2),
            frameSize = (-width/6.0, width/6.0,
                         base.a2dTop, base.a2dBottom),
            frameColor = (0,0,0,0))
        self.charSelectFrame.reparentTo(self.frameMain)

        self.btnChar1 = self.createCharacterButton(
            (-0.2, 0, 0),
            "assets/gui/Char1Button.png",
            1)
        self.btnChar1.reparentTo(self.charSelectFrame)

        self.btnChar2 = self.createCharacterButton(
            (0.2, 0, 0),
            "assets/gui/Char2Button.png",
            2)
        self.btnChar2.reparentTo(self.charSelectFrame)

        self.btnBack = DirectButton(
            text = "BACK",
            text_fg = (1,1,1,1),
            text_align = TextNode.ALeft,
            scale = 0.1,
            pad = (0.15, 0.15),
```

```
            pos = (base.a2dLeft + 0.08, 0, -0.03),
            frameColor = (
                (0.2,0.2,0.2,0.8),
                (0.4,0.4,0.4,0.8),
                (0.4,0.4,0.4,0.8),
                (0.1,0.1,0.1,0.8)),
            relief = 1,
            command = base.messenger.send,
            extraArgs = ["CharSelection-Back"],
            pressEffect = False,
            rolloverSound = None,
            clickSound = None)
        self.btnBack.setTransparency(1)
        self.btnBack.reparentTo(self.footerFrame)

        self.btnStart = DirectButton(
            text = "START",
            text_fg = (1,1,1,1),
            text_align = TextNode.ARight,
            scale = 0.1,
            pad = (0.15, 0.15),
            pos = (base.a2dRight - 0.08, 0, -0.03),
            relief = 1,
            frameColor = (
                (0.2,0.2,0.2,0.8),
                (0.4,0.4,0.4,0.8),
                (0.4,0.4,0.4,0.8),
                (0.1,0.1,0.1,0.8)),
            command = base.messenger.send,
            extraArgs = ["CharSelection-Start"],
            pressEffect = False,
            rolloverSound = None,
            clickSound = None)
        self.btnStart.setTransparency(1)
        self.btnStart.reparentTo(self.footerFrame)
        self.btnStart["state"] = DGG.DISABLED

        self.hide()

    def createCharacterButton(self, pos, image, charNr):
        btn = DirectButton(
            scale = 0.1,
            relief = 0,
            frameColor = (0,0,0,0),
            pos = pos,
            image = image,
            command = self.selectCharacter,
            extraArgs = [charNr],
            rolloverSound = None,
            clickSound = None)
        btn.setTransparency(1)
        return btn

    def selectCharacter(self, charNr):
        if self.char1Frame["image"] == None:
            self.char1Frame["image"] = "assets/gui/Char%d_L.png" % charNr
            self.char1Frame["image_scale"] = (0.5,1, 1)
            self.selectedCharacter1 = charNr
```

```
            self.footerFrame["text"] = "PLAYER 2 - CHOOSE YOUR CHARACTER"
        elif self.char2Frame["image"] == None:
            self.char2Frame["image"] = "assets/gui/Char%d_R.png" % charNr
            self.char2Frame["image_scale"] = (0.5,1, 1)
            self.selectedCharacter2 = charNr
            self.btnStart["state"] = DGG.NORMAL
            self.footerFrame["text"] = "START THE FIGHT >"

    def show(self):
        self.selectedCharacter1 = None
        self.selectedCharacter2 = None
        self.char1Frame["image"] = None
        self.char2Frame["image"] = None
        self.footerFrame["text"] = "PLAYER 1 - CHOOSE YOUR CHARACTER"
        self.btnStart["state"] = DGG.DISABLED
        self.frameMain.show()

    def hide(self):
        self.frameMain.hide()
```

It's quite lengthy but most of the source should already be known to you as it is either basic python code or we already used it in the other menu. So we will only highlight and go over the new parts in this menu.

The first new thing we learn here is how we can texturing elements in Panda3D. Aside of images as we learned earlier we can also assign textures to the DirectGUI elements which we will use here as it gives us a bit more control of how the element should be styled. The texture we create here could also be used for any other node you have set up within your Panda3D application, there is not too much different from texturing DirectGUI elements to 3D Models.

Another thing you might also notice is how we have structured the parenting of the objects. We still have one big main frame, but also a few smaller frames which we reparent other GUI elements to. These elements still will be hidden as soon as the main frame gets hidden. This will be propagated down the complete tree of children of the main frame.

Also we set the set a state of the button manually the first time with

```
self.btnStart["state"] = DGG.DISABLED
```

This will set the button to a state, where the player can't click the button and it'll show the 4^{th} state, hence the (0.1,0.1,0.1,0.8) frame color we set.

Lets take a look at the functions we have set up here.

The createCharacterButton will help us create buttons for each selectable character in the game. As every character button will be styled the same we just need to pass it the character specific parameters as the buttons position, the characters image which should be put on the button and finally the characters number which we will then use later to determine which character the players have selected; hence we will send this number with the event we send on the button click.

Next comes the function which we call on a click of one of the the buttons. In it we check if the left and right frames already have a character image assigned and if not, assign one to them. First we let Player 1 decide which character he wants to play and secondly Player 2. As soon as both have made a decision, we enable the start button so they can proceed to the next part. Also we set the footer frames text to give the players a hint on what they should do next.

Finally we come to the usual show and hide functions. Though we also use the show function to reset the menu so whenever the player moves away and then back to the menu it will have no characters selected the text reset to the selection for the first character and the start button disabled again.

Having the menu done, we need to enable it to be used from within the main file and change some things of our earlier character setup, so lets first add and change the following things.

```
…
from menu import Menu
from characterselection import CharacterSelection

…

        self.menu = Menu()
        self.charSelection = CharacterSelection()
        …

    #
    # FSM PART
    #
    def enterMenu(self):
        self.accept("Menu-Start", self.request, ["CharSelection"])

    …

    def enterCharSelection(self):
        self.accept("CharSelection-Back", self.request, ["Menu"])
        self.accept("CharSelection-Start", self.request, ["Game"])
        self.charSelection.show()

    def exitCharSelection(self):
        self.ignore("CharSelection-Start")
        self.ignore("CharSelection-Back")
        self.charSelection.hide()
        self.selectedChar1 = self.charSelection.selectedCharacter1
        self.selectedChar2 = self.charSelection.selectedCharacter2

    def enterGame(self):
        # main game code should be called here
        self.arena = Arena(1)
        self.arena.start()
        self.camera.setPos(0, -5, 1.25)
        self.player = Player(0, self.selectedChar1, "p1")
        self.player2 = Player(1, self.selectedChar2, "p2")
        self.player.setEnemy(self.player2.collisionNodeName)
        self.player2.setEnemy(self.player.collisionNodeName)
        self.player.start(self.arena.getStartPos(1))
```

```
        self.player2.start(self.arena.getStartPos(2))
        self.taskMgr.add(self.updateWorldCam, "world camera update task")
        …
```

That's to the additions, now as we've put the character initialization, we need to remove it from the postInit method of the main file.

The here red marked lines need to be removed from the main file's postInit method.

```
        self.arena.start()
        self.camera.setPos(0, -5, 1.25)
        self.player.start(self.arena.getStartPos(1))
        self.player2.start(self.arena.getStartPos(2))
        self.player.setEnemy(self.player2.collisionNodeName)
        self.player2.setEnemy(self.player.collisionNodeName)
```

After removing them, we need to change the characters stop method to cleanly remove it. So change the lines of the stop method in the player class tot he following.

```
    def stop(self):
        self.ignoreAll()
        taskMgr.remove("move task %d"%self.charId)
        self.character.cleanup()
        self.character.removeNode()
```

Now you should be able to rerun your game and get into the character selection screen as soon as you hit the start button within the main menu as well as start a fight as soon as you have selected two characters and hit the start button in the character selection screen.

Now let's move on to the next menu we will have in line, the Level selection.

Level selection

The level selection will gives the players the chance to select between a few available arenas to fight in. It will be set up almost the same way as the previous character selection screen. This menu will be set between the character selection and the start of the fighting.

As with the character selection, we will start again with a new file, this time called levelselection.py Within this file, add the following code, which will set up the menu quite similar to the character selection but with just the levels as buttons and a back button.

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

from panda3d.core import TextNode
from direct.gui.DirectGui import (
    DirectFrame,
    DirectButton)

class LevelSelection:
    def __init__(self):

        self.frameMain = DirectFrame(
            frameSize = (base.a2dLeft, base.a2dRight,
                         base.a2dTop, base.a2dBottom),
            frameColor = (0.05, 0.05, 0.05, 1))
        self.frameMain.setTransparency(1)

        self.btnLevel1 = self.createLevelButton(
            (-0.6, 0, 0.15),
            "assets/gui/Level1Button.png",
            0)
        self.btnLevel1.reparentTo(self.frameMain)

        self.btnLevel2 = self.createLevelButton(
            (0.6, 0, 0.15),
            "assets/gui/Level2Button.png",
            1)
        self.btnLevel2.reparentTo(self.frameMain)

        self.footerFrame = DirectFrame(
            text = "SELECT THE ARENA",
            text_fg = (1,1,1,1),
            text_scale = 0.08,
            text_pos = (0, -0.03),
            frameSize = (base.a2dLeft, base.a2dRight,
                         -0.1, 0.1),
            pos = (0, 0, base.a2dBottom + 0.2),
            frameColor = (0, 0, 0, 0.5))
        self.footerFrame.setTransparency(1)
        self.footerFrame.reparentTo(self.frameMain)
```

```
        self.btnBack = DirectButton(
            text = "BACK",
            text_fg = (1,1,1,1),
            text_align = TextNode.ALeft,
            scale = 0.1,
            pad = (0.15, 0.15),
            pos = (base.a2dLeft + 0.08, 0, -0.03),
            frameColor = (
                (0.2,0.2,0.2,0.8),
                (0.4,0.4,0.4,0.8),
                (0.4,0.4,0.4,0.8),
                (0.1,0.1,0.1,0.8),
                ),
            relief = 1,
            command = base.messenger.send,
            extraArgs = ["LevelSelection-Back"],
            pressEffect = False,
            rolloverSound = None,
            clickSound = None)
        self.btnBack.setTransparency(1)
        self.btnBack.reparentTo(self.footerFrame)

        self.hide()

    def createLevelButton(self, pos, image, levelNr):
        self.selectedLevel = levelNr
        btn = DirectButton(
            scale = (0.5, 1, 0.75),
            relief = 0,
            frameColor = (0,0,0,0),
            pos = pos,
            image = image,
            command = base.messenger.send,
            extraArgs = ["LevelSelection-Start"],
            rolloverSound = None,
            clickSound = None)
        btn.setTransparency(1)
        return btn

    def show(self):
        self.frameMain.show()

    def hide(self):
        self.frameMain.hide()
```

Everything in this menu should already be known to you as it's only a kind of stripped down character selection so we will move right on to the integration in our main script.

```
from characterselection import CharacterSelection
from levelselection import LevelSelection
…
        self.charSelection = CharacterSelection()
        self.levelSelection = LevelSelection()
    …
    def enterCharSelection(self):
```

```
        self.accept("CharSelection-Back", self.request, ["Menu"])
        self.accept("CharSelection-Start", self.request, ["LevelSelection"])
    …

    def enterLevelSelection(self):
        self.accept("LevelSelection-Back", self.request, ["CharSelection"])
        self.accept("LevelSelection-Start", self.request, ["Game"])
        self.levelSelection.show()

    def exitLevelSelection(self):
        self.ignore("LevelSelection-Start")
        self.ignore("LevelSelection-Back")
        self.levelSelection.hide()

    …

    def __escape(self):
        if self.state == "Menu":
            self.quit()
        elif self.state == "LevelSelection":
            self.request("CharSelection")
        else:
            self.request("Menu")
```

That's it, with this you now have a functional simple level selection screen which will show up directly after the characters have been selected.

Now we have all menus done we want to have in this Tutorial, following we will show you a few things of how you can use DirectGUI to enhance the main game part look and feel.

K.O. screen

The first thing we will enter into the game which uses the DirectGUI but isn't really a menu but more an informational screen after the fight is over is the K.O. screen. It is a really simple screen where we just use the basics we already know but create an important part which enhances the gameplay and look and feel of the game. This screen should also show the price the players were fighting for to give the winner a bit of satisfaction. For this screen, just add the following code in a new file called koscreen.py.

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

# Panda3D imoprts
from direct.showbase.DirectObject import DirectObject
from direct.gui.DirectGui import (
    DirectFrame,
    DirectLabel,
    DirectButton)

class KoScreen(DirectObject):
    def __init__(self):
        self.frameMain = DirectFrame(
            frameSize = (base.a2dLeft, base.a2dRight,
                         base.a2dTop, base.a2dBottom),
            frameColor = (0, 0, 0, 0.75))
        self.frameMain.setTransparency(1)

        self.lbl_KO = DirectLabel(
            text = "K.O.",
            text_fg = (1,1,1,1),
            scale = 1,
            pos = (0, 0, 0),
            frameColor = (0,0,0,0))
        self.lbl_KO.setTransparency(1)
        self.lbl_KO.reparentTo(self.frameMain)

        self.lbl_PlayerXWon = DirectLabel(
            text = "PLAYER X WON",
            text_fg = (1,1,1,1),
            scale = 0.25,
            pos = (0, 0, -0.5),
            frameColor = (0,0,0,0))
        self.lbl_PlayerXWon.setTransparency(1)
        self.lbl_PlayerXWon.reparentTo(self.frameMain)

        self.btnContinue = DirectButton(
            text = "CONTINUE",
            text_fg = (1,1,1,1),
            scale = 0.1,
            pad = (0.15, 0.15),
```

```
            pos = (0, 0, -0.8),
            frameColor = (
                (0.2,0.2,0.2,0.8),
                (0.4,0.4,0.4,0.8),
                (0.4,0.4,0.4,0.8),
                (0.1,0.1,0.1,0.8),
                ),
            relief = 1,
            command = base.messenger.send,
            extraArgs = ["KoScreen-Back"],
            pressEffect = False,
            rolloverSound = None,
            clickSound = None)
        self.btnContinue.setTransparency(1)
        self.btnContinue.reparentTo(self.frameMain)

        self.hide()

    def show(self, succseedingPlayer):
        self.frameMain.show()
        self.lbl_PlayerXWon["text"] = "PLAYER %d WON" % succseedingPlayer

    def hide(self):
        self.frameMain.hide()
```

Now as we want this to be visible after the game's over but still have the level and characters in the background we can't add it as another state to our FSM. We will instead add it as a simple function which will get called as soon as the gameOver event got thrown by any of the characters.

```
from characterselection import CharacterSelection
from levelselection import LevelSelection
from koscreen import KoScreen

…

        self.levelSelection = LevelSelection()
        self.koScreen = KoScreen()
        …

    def enterMenu(self):
        self.accept("Menu-Start", self.request, ["CharSelection"])
        self.accept("Menu-Quit", self.quit)
        self.ignore("KoScreen-Back")
        self.koScreen.hide()

        …

        self.player2.start(self.arena.getStartPos(2))
        self.taskMgr.add(self.updateWorldCam, "world camera update task")
        self.accept("gameOver", self.gameOver)
        …

        del self.player2
        self.arena.stop()
        self.ignore("gameOver")
```

```
    #
    # BASIC FUNCTIONS
    #
    def gameOver(self, LoosingCharId):
        winningChar = 1
        if LoosingCharId == 0:
            winningChar = 2
        self.accept("KoScreen-Back", self.request, ["Menu"])
        self.koScreen.show(winningChar)
```

Having that we will have a K.O. screen that get's displayed when a character got defeated and hitting the continue button in that screen will return to the main menu.

HUD Head-up-Display

Having menus is fine, though, the DirectGUI framework can also be used to create Head-up-Displays. These can have various functionalities and look and feel. A HUD can be quite statically put as 2D images on the screen which was mostly the case in old games and currently still is useful for things like maps and story text/subtitles as well as cross-hairs. Though, modern games especially shooters tend to put more and more HUD related parts directly into the 3D space which gives it a kind of Artificial Reality (AR) typical appearance.

For our game, we take it simple and just add two bars which shows the characters health as well as an image of the character aside of that bars.

The HUD will have its own script so lets start by adding a new file called hud.py in the usual games source directory.

Within this file, add the following code which will set up the HUD for us.

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

# Panda3D imoprts
from direct.showbase.DirectObject import DirectObject
from direct.gui.DirectGui import DirectWaitBar, DGG
from panda3d.core import TextNode

class Hud(DirectObject):
    def __init__(self):
        self.lifeBar1 = DirectWaitBar(
            text = "Player1",
            text_fg = (1,1,1,1),
            text_pos = (-1.2, -0.18, 0),
            text_align = TextNode.ALeft,
            value = 100,
            barColor = (0, 1, 0.25, 1),
```

```
            barRelief = DGG.RAISED,
            barBorderWidth = (0.03, 0.03),
            borderWidth = (0.01, 0.01),
            relief = DGG.RIDGE,
            frameColor = (0.8,0.05,0.10,1),
            frameSize = (-1.2, 0, -0.1, 0),
            pos = (-0.2,0,base.a2dTop-0.15))
        self.lifeBar1.setTransparency(1)

        self.lifeBar2 = DirectWaitBar(
            text = "Player2",
            text_fg = (1,1,1,1),
            text_pos = (1.2, -0.18, 0),
            text_align = TextNode.ARight,
            value = 100,
            barColor = (0, 1, 0.25, 1),
            barRelief = DGG.RAISED,
            barBorderWidth = (0.03, 0.03),
            borderWidth = (0.01, 0.01),
            relief = DGG.RIDGE,
            frameColor = (0.8,0.05,0.10,1),
            frameSize = (0, 1.2, -0.1, 0),
            pos = (0.2,0,base.a2dTop-0.15))
        self.lifeBar2.setTransparency(1)

        self.accept("hud_setLifeBarValue", self.setLifeBarValue)
        self.hide()

    def show(self):
        self.lifeBar1["value"] = 100
        self.lifeBar2["value"] = 100
        self.lifeBar1.show()
        self.lifeBar2.show()

    def hide(self):
        self.lifeBar1.hide()
        self.lifeBar2.hide()

    def setLifeBarValue(self, barNr, newValue):
        if barNr == 0:
            self.lifeBar1["value"] = newValue
        elif barNr == 1:
            self.lifeBar2["value"] = newValue
```

Lets go through this and see what's new to what we already learned in earlier sections. The main new thing here is the DirectWaitBar and the many parameters we pass to it, which is the way how DirectGUI elements are styled and set up. You may wonder why we use a wait bar which would normally be used as the name already states as a wait bar for loading screens. The reason is quite simple, it gives us all the functionality we need for other bars like our lifebars we need for the players, so just don't let the name confuse you and use whatever suits you best even if the name would assume another thing. But now lets take a closer look at those parameters.

```
text = "Player1",
text_fg = (1,1,1,1),
```

```
text_pos = (-1.2, -0.18, 0),
text_align = TextNode.ALeft,
```

The first four parameters describe the text node of the DirectWaitBar, we will use it to make clear which bar belongs to which player. The text will be placed below the bar at the left or right side of the bar. Also we let the text be drawn in a white color and have it's alignment be respective to the side of the screen it is placed. For the alignment we import a new base class of Panda3D namely TextNode. The TextNode implements most of the features and functions you would need for placing and setting up text in your application. You can also use just a TextNode to add text to your screen, though, there are better suited classes like OnscreenText and DirectLabel if you want to add text.

```
value = 100,
barColor = (0, 1, 0.25, 1),
barRelief = DGG.RAISED,
barBorderWidth = (0.03, 0.03),
```

Here we have the set up and styling of our bar, to be precisely, the green part of our lifebar that will actually get changed. The value determines the initial and actual amount the bar is showing by default the bars range is from 0 to 100 and value default is set to 0. But as we have a lifebar and start with 100% healthy characters, we change that to 100 to be able to count down with that. Next we set the bars color to a slightly blue green color. The barRelief determines the style of how the bar gets drawn RAISED will let the bar appear to pop out of the screen means, adding lighting on the top and left border of it and shadow on the bottom and right border. This border can then be changed with the next parameter barBorderWidth. This expects a two value tuple with the usual Panda3D 3D-space units. The first value is used for left and right and the second for the bottom and top border.

Note, DGG stands for Direct GUI Globals and contains a few useful definitions that will often be used for defining DirectGUI styles.

```
borderWidth = (0.01, 0.01),
relief = DGG.RIDGE,
frameColor = (0.8,0.05,0.10,1),
frameSize = (-1.2, 0, -0.1, 0),
```

With this four parameters, we control the background frame of our lifebar. We begin with the border which is just the same as above for the bar itself, just this time for the background. Note that the border will be added around the frame and hence increase its size to not change it's inside elements. For this border we use a RIDGE style that represents a kind of wall like border. Finally we set the color to a dark red and change the frames size which will change the 0-point of the whole bar. This will on the other hand help us with placing the bar, as the 0-point is now on the top right side of the bar.

```
pos = (-0.2,0,base.a2dTop-0.15)
```

Finally we set the positions of the bars to be on the top left and right side of the screen. As we've done that quite often already with other elements we won't describe this any further.

That's all about styling for now, with those informations, you should be able to also style most of the other DirectGUI elements. You should also take a look at the source code of the DirectGUI elements if you want to use them to find out any configurations you can do on them and check on what they are based.

Following to the setup of the bars, we also catch an event that we will use to catch requests for life changes of the characters to change the bars value.

Taking a look at the function called for that event you may notice the following line.

```
self.lifeBar1["value"] = newValue
```

This is a nice functionality of DirectGUI elements using a bit of python magic. Almost all parameters which were set at initialization of an element can later be interactively changed by using the name of the parameter as index to the GUI element and simply assign the new value to it. Most of the time the new value will cause the element to be automatically be updated to also display the new value.

Finally as we now have the class for our HUD set up, we need to use it in our application, so add the following lines to the respective classes.

In the Player class add

```
        self.health -= 10
        base.messenger.send(
            "lifeChanged",
            [self.charId, self.health])
        if self.health <= 0:
            self.gotDefeated = True
```

This makes sure the HUD will always get updated whenever one of the players health changes. Note, that we use another event name as we catch in the HUD. This will make our classes more interchangeable with other applications as you may not always have a HUD but want to catch life changes or general events for other things too. Hence we throw Player specific events from within the player which will then be caught in the glue code we will ad to our Main class to “glue” the player to the HUD. The glue code you have to add to Main looks as follow and has to be added to the bottom of the enterGame method.

```
        self.hud.show()
        def lifeChanged(charId, health):
            base.messenger.send(
                "hud_setLifeBarValue",
                [charId, health])
        self.accept("lifeChanged", lifeChanged)
```

Here we define a small function, which will simply forward the event as thrown by the player to the HUD. This method could possibly be extended with anything that needs to know if a player got hit but isn't part of the player class itself. To have it clean up in the end again, you also need to add this to the exitGame method.

```
        self.hud.hide()
        self.ignore("lifeChanged")
```

Finally, as without that the whole HUD thing won't work as we set it up till now, we need to import and instantiate our HUD.

```
…
from menu import Menu
from characterselection import CharacterSelection
from hud import Hud
…
        self.charSelection = CharacterSelection()
        self.charSelection.show()
        self.hud = Hud()

        #
        # Event handling
        …
```

Now we have a nice looking, functional and simply made HUD. If you test your game again, you should see the HUD when you start a fight as well as notice the changes to the bars whenever a player got hit by the other player.

Credits

The last menu part we will add to our game is the credits section which we will use to list all authors of the game. Even though this may just be useful for games with a few authors, you may also add this if it's just you alone who has written the game and add some other information to it like license information and if applicable honor and mention other contributors or external works that have been used in the game.

To add a credits section in our game, we will first create another file called credits.py which then can be used to show the credits either from the main menu, at the end of the game or wherever else you want to show them. This class we will create will also be a rather simple one so we just show the code here and describe the new parts we use in it.

```
#!/usr/bin/python
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

from panda3d.core import (
    TextNode,
    TextProperties,
    TextPropertiesManager)
from direct.gui.DirectGui import (
    DirectFrame,
    DirectLabel,
    DirectButton)
from direct.stdpy.file import open

class Credits:
    def __init__(self):

        self.frameMain = DirectFrame(
            frameSize = (base.a2dLeft, base.a2dRight,
                         base.a2dTop, base.a2dBottom),
            frameColor = (0.05, 0.05, 0.05, 1))
        self.frameMain.setTransparency(1)

        tpBig = TextProperties()
        tpBig.setTextScale(1.5)
        tpSmall = TextProperties()
        tpSmall.setTextScale(0.75)
        tpUs = TextProperties()
        tpUs.setUnderscore(True)
        tpMgr = TextPropertiesManager.getGlobalPtr()
        tpMgr.setProperties("big", tpBig)
        tpMgr.setProperties("small", tpSmall)
        tpMgr.setProperties("us", tpUs)

        creditsText = ""
        with open("credits.txt") as f:
            creditsText = f.read()
```

```
        self.lblCredits = DirectLabel(
            text = creditsText,
            text_fg = (1,1,1,1),
            text_bg = (0,0,0,0),
            frameColor = (0,0,0,0),
            text_align = TextNode.ACenter,
            scale = 0.1,
            pos = (0, 0, base.a2dTop - 0.2))
        self.lblCredits.setTransparency(1)
        self.lblCredits.reparentTo(self.frameMain)

        self.creditsScroll = LerpPosInterval(
            self.lblCredits,
            12.0,
            (0, 0, base.a2dTop + 3.5),
            startPos=(0, 0, base.a2dBottom),
            name="CreditsScroll")

        self.btnBack = DirectButton(
            text = "BACK",
            text_fg = (1,1,1,1),
            text_align = TextNode.ALeft,
            scale = 0.1,
            pad = (0.15, 0.15),
            pos = (base.a2dLeft + 0.08, 0, base.a2dBottom + 0.05),
            frameColor = (
                (0.2,0.2,0.2,0.8),
                (0.4,0.4,0.4,0.8),
                (0.4,0.4,0.4,0.8),
                (0.1,0.1,0.1,0.8),
                ),
            relief = 1,
            command = base.messenger.send,
            extraArgs = ["Credits-Back"],
            pressEffect = False,
            rolloverSound = None,
            clickSound = None)
        self.btnBack.setTransparency(1)
        self.btnBack.reparentTo(self.frameMain)

        self.hide()

    def show(self):
        self.frameMain.show()
        self.creditsScroll.loop()

    def hide(self):
        self.frameMain.hide()
        self.creditsScroll.finish()
```

The first new thing you might notice is that we import a replacement for pythons file handling. This replacement is pandas thread safe implementation and should be used whenever files are loaded within a Panda3D application. Later on in the class, we use the new file handling from Panda3D just as we would do with the default python file loading.

The next new feature we have in this class are the TextProperties. Those can be used in any text that

will be created and rendered by Panda3D. You can set up any kind of text property and also combine them. To use those properties in our texts, we will add a formation just like this one:

```
\1big\1Tatakai no ikimono\2
```

whereas the \1 and \2 are special characters and need to be replaced by their correct character if we want to load them from a file and not add them directly within a python script. Just take a look at the credits.txt file to get the special characters out of it. You can also write a simple print statement with the \1 and \2 in place which will render the correct signs in the terminal.

As you also may see, we use a LerpPosInterval. This is a function that can be used to move around any nodes we hand over to it. There also are other Lerp intervals for other values like rotations. We use this interval to let the credits text scroll over the screen, hence we give it the credits text node, followed by the time the interval should take to move the credits from one point to the other, the destination and start position as well as a name for the interval. In the show and hide functions of the class we also start and stop the interval.

Now we just need to add it to the usual places to get it into our main menu.

In main.py add the following parts

```
…
from menu import Menu
from credits import Credits
from characterselection import CharacterSelection
…


        …
        self.menu = Menu()
        self.credits = Credits()
        self.charSelection = CharacterSelection()
        …

    def enterMenu(self):
        show_cursor()
        self.accept("Menu-Start", self.request, ["CharSelection"])
        self.accept("Menu-Credits", self.request, ["Credits"])
        self.accept("Menu-Quit", self.quit)
        …

    def exitMenu(self):
        self.ignore("Menu-Start")
        self.ignore("Menu-Credits")
        self.ignore("Menu-Quit")
        self.menu.hide()

    def enterCredits(self):
        self.accept("Credits-Back", self.request, ["Menu"])
        self.credits.show()

    def exitCredits(self):
        self.ignore("Credits-Back")
```

```
        self.credits.hide()
```

Now we need to enhance our main menu to get a new button for the credits in there. To achieve that, add and change the following lines to the menu.py scripts class.

```
        self.btnStart = self.createButton(
            "Start",
            -.10,
            ["Menu-Start"])

        self.btnStart = self.createButton(
            "Credits",
            -.25,
            ["Menu-Credits"])

        self.btnExit = self.createButton(
           "Quit",
            -.40,
            ["Menu-Quit"])
```

That's all we need to do for the credits, now you should see a new button in the main menu which will revile the credits on click. The credits should render the text as given in the credits.txt file with it's special formating set and a back button to be able to return to the main menu.

If you like you can also set the credits to be the menu where the player lands after a fight. You can do that by changing the following line in the gameOver function

```
self.accept("KoScreen-Back", self.request, ["Credits"])
```

as well as add the self.koScreen.hide() function call to the enterCredits function to hide the KO screen there as well.

From here on, we have anything a simple game needs to be quite enjoyable to play in the next sections we will take a look at making our game more appealing to the users by adding a few more fancy graphic additions as well as some sound.

Game Enhancements

From here on we have the most important gameplay elements done and from now on will just have a few nice to have enhancements that makes the game look more finished. It will contain things like cursor changes and audio playback as well as some other graphic enhancements.

Panda3D will give you some easy to use functions that will give you some simple but good looking enhancements rather fast using the shader generator and other additions that can simply be switched on and off. Though they may not be the most performant ways they sure are the most simple ones. If you want to have performance rather than simplicity you may need to write more complex and specialized versions of the specific things you need.

Mouse cursor

The first enhancement we will make is changing the mouse cursor as well as hiding it in the right situations, namely when entering the fight and showing it again as soon as the fight is over.

For that we first create a little helper script with two functions to show and hide the cursor as well as set the cursor within the show function. So create a file called helper.py and add the following code to it.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
__author__ = "Fireclaw the Fox"
__license__ = """
Simplified BSD (BSD 2-Clause) License.
See License.txt or http://opensource.org/licenses/BSD-2-Clause for more info
"""

#
# PYTHON IMPORTS
#
import sys

#
# PANDA3D ENGINE IMPORTS
#
from pandac.PandaModules import WindowProperties

def hide_cursor():
    """set the Cursor invisible"""
    props = WindowProperties()
    props.setCursorHidden(True)
    # somehow the window gets undecorated after hiding the cursor
    # so we reset it here to the value we need
    #props.setUndecorated(settings.fullscreen)
    base.win.requestProperties(props)
```

```
def show_cursor():
    """set the Cursor visible again"""
    props = WindowProperties()
    props.setCursorHidden(False)
    # set the filename to the mouse cursor
    x11 = "assets/gui/Cursor.x11"
    win = "assets/gui/Cursor.ico"
    if sys.platform.startswith("linux"):
        props.setCursorFilename(x11)
    else:
        props.setCursorFilename(win)
    base.win.requestProperties(props)
```

Here we learn how to use the Panda3D window properties. Those porperties are not only useful for changing the cursor for the application but also the window size, if it should be fullscreen and enabling and disabling the windows decoration as well as a few other application window specific settings.
Note that for the cursor to be usable on all platforms, we need to have at least two cursor types, one for X11 window systems like Linux and Mac and another one for Windows systems.

Now we will use it within our main file by importing the functions and call them at the right places.

```
from hud import Hud
from helper import hide_cursor, show_cursor
…
    #
    # FSM PART
    #
    def enterMenu(self):
        show_cursor()

    …

        def lifeChanged(charId, health):
            base.messenger.send(
                "hud_setLifeBarValue",
                [charId, health])
        self.accept("lifeChanged", lifeChanged)
        hide_cursor()

    …

    #
    # BASIC FUNCTIONS
    #
    def gameOver(self, LoosingCharId):
        show_cursor()
```

That's all to show the new cursor and hide and show it correct.

Audio

This section will teach you another quite big and important section in game development. It will talk about the various kinds of audio you can find in games. Imagine a game without it's sound, it may still be playable but it will be not the same as with audio. The sound of a game adds really much to the mood of a game. Some games may even be harder or impossible to play like dance dance revolution or Rhythm Thief.

Even if you don't have such a music related game, some good sound effects (abbreviated as SFX) will add a lot to your game, not just to the mood but also to the realism of the game. Just imagine a big explosion which looks good but don't have any boom sound, it will just be half as awesome as it would be with sound. On the other hand wrongly placed soundtracks might confuse the player, for example if a footstep sound is played but no one's walking. This also is the case for music and the games mood, for example if you have a bright scenery but have some scary horror music in the background. So always make sure, the sound fits to the desired result of the game.

Now, if you don't have some musical skills and no one who can create some music and other necessary audio for you, you can find good audio files on the Internet on a variety of websites like https://www.jamendo.com/ and https://soundcloud.com/. If it's sound effects that you need, a good source on the Internet is the https://freesound.org/ website.

As a side note for programmers, Panda3D comes with a few different audio processing libraries and will use OpenAL as default. If you want to use some special effects like DSP, you either need to switch to the other by Panda3D supported audio library named FMOD, though keep in mind that you'll need to purchase a license for that if your application will be a commercial one. Otherwise you need to search for another alternative to add effects and implement that by yourself.

Music

In the first part of the Audio section we will talk about music creation and how to get it into your Panda3D application.

Creating music

To create music, there are many ways which range from playing and recording music with real instruments till using computer generated midi sound files that emulate the real world instruments. This tutorial will give a quick view into the world of music creation, it will not teach you music theory.

The first thing you might get into while learning music creation in general are score editors and general music trackers like TuxGuitar, Rosegarden and MusE. Those can be used to write down notes and listen to them right away. There also are simple but complete solutions for audio creation and editing like the Linux Multimedia Studio short LMMS. Dependent on the soundfonts and audio samples which will be used to generate the audio output of such software you might also be able to

export or record the music directly from those trackers and not have to record from a live orchestra or band.

Now, when you like to record from TuxGuitar, you need to "convert" your exported midi file to an audio file like flac or wav which then can be worked with in an audio editor like audacity. The applications which creates audio files out of midi files are called software synthesizers like TiMidity. Using audacity or any other audio editing tools, you can add effects to the music or cut and reassemble it in different versions. Note, you should use a new track for each instrument you have in your song as this will give you more power over editing the song you have.

If you have an orchestra or band behind you which will be able to write and play music for you, they should also have the required knowledge of recording the song or at least get to a studio and let them record your music. If you want to record by yourself, you should take a look at more professional audio recording software solutions like Ardour in combination with Jack as input and output routing backend.

Bring the music into the game

After you found or made some nice soundtracks for your game, you need to bring it in. Loading music is just as simple, if not even a bit more simple than loading 3D models into a scene. You can simply call myMusic = loader.loadMusic("path/to/music.ogg") and afterwards simply call myMusic.play() to start the music playback. Note, by default Panda3D will only play one music file at a time, if you want to play more than one music simultaneously you need to change the following configuration variable within a config.prc file.

Also note, that if you use the main.py template form the panda3d-code-collection repository you will have a specific configuration variables that you can add to the games specific .prc file namely audio-mute. This variable if set to #t or 1 will mute all all audio in the game, SFX as well as Music. If you want to use the configuration file to disable SFX or Music specifically, you can use Panda3Ds default configuration variables audio-music-active and audio-sfx-active. Also there are specific variables for changing the volume. For a list of those variables just take a look at the http://www.panda3d.org/manual/index.php/List_of_All_Config_Variables manual page.

For our tutorial we will add the following lines of code into our Main class to play two music files, one will be for the menus and another one for the fight.

```
        …
        self.levelSelection = LevelSelection()
        self.koScreen = KoScreen()
        self.hud = Hud()
        self.menuMusic = loader.loadMusic("assets/audio/menuMusic.ogg")
        self.menuMusic.setLoop(True)
        self.fightMusic = loader.loadMusic("assets/audio/fightMusic.ogg")
        self.fightMusic.setLoop(True)

        …
```

```
        self.ignore("KoScreen-Back")
        self.koScreen.hide()
        self.menu.show()
        if self.menuMusic.status() != AudioSound.PLAYING:
            self.menuMusic.play()
        if self.fightMusic.status() == AudioSound.PLAYING:
            self.fightMusic.stop()

        …

        def lifeChanged(charId, health):
            base.messenger.send(
                "hud_setLifeBarValue",
                [charId, health])
        self.accept("lifeChanged", lifeChanged)
        hide_cursor()
        if self.fightMusic.status() != AudioSound.PLAYING:
            self.fightMusic.play()
        if self.menuMusic.status() == AudioSound.PLAYING:
            self.menuMusic.stop()
```

The first two lines will load our music just like we've done it with models the only difference is that we use the loadMusic function of the loader class.

Later when we want to use it we first check the sounds state. There are tree states that we can check for which are READY, BAD and PLAYING whereas ready and bad are useful to check if the sound got loaded correctly and can be played back where playing determines if the sound is currently active. In our code, we use the playing state to check if the determined music file is already playing and if not start or stop it respectively. Unlike the Actor class an audio sound does not have a loop function but only a play method that will play the sound. If you want to have the sound played in a loop, you need to set it via the setLoop function.

You can also change the audios volume and panning with functions directly on the music variable if you want to have each music at a different pan and volume but you can also set the overall music volume using the audio manager which is stored in the base.musicManager variable. This manager can also be used to disable all music of the application. For a little more detailed description on the audioManagers, take a look at this manual page
http://www.panda3d.org/manual/index.php/Audio_Managers

Ambient sounds

Aside of Music, ambient sounds are the next best thing to greatly define the mood and realism of a game. Ambient sounds are things like the birds tweeting, wind blowing through the trees and crowds in the background.

We will load the ambient sounds, which we will also store in our assets/sounds/ directory, within our level class as ambient sounds most of the time are level specific and hence are best placed there. For the ambient sound loading we will use another audio specific loader function namely loadSfx. This function does exactly the same as the loadMusic function with the only difference, that it loads the sound file to another list, a list within the base.sfxManager. Hence the overall configurations for SFX has to be done with that specific manager. Panda3D can by default have 16 different sounds cached at a given time for each manager. If you need more you need to change it in a config.prc file with the audio-cache-limit.

Now, to get some ambient sounds in our level, we will add almost the same as we have done for the music loading in our Main class but this time in the Arena class. So add the following lines of code in the arena.py script.

```
        self.sunNp.setPos(-10, 10, 30)
        self.sunNp.lookAt(0,0,0)

        self.ambientSound = None
        if arenaNr == 1:
            self.ambientSound = loader.loadSfx("assets/audio/ambientLevel1.ogg")
            self.ambientSound.setLoop(True)

        …

        render.setLight(self.alnp)
        render.setLight(self.sunNp)
        if self.ambientSound != None:
            self.ambientSound.play()

        …

        self.arena.hide()
        render.clearLight()
        if self.ambientSound != None:
            self.ambientSound.stop()
```

As you see, we first load the sound with the loadSfx from our assets/audio/ directory and then simply start and stop the audio track within the respective start and stop functions of the arena Class.

SFX

Finally we have the sound effects or for short, SFX. Those are things like footsteps, punch hits, bullet shots, explosions or click sounds a switch would make when activated. Most of the time those effects don't really represent the real world sound but are a much more pregnant version of it which is the case most games and also movies have sound designers specialized at finding good sounding sounds that fit to the action.

We already used the loadSFX function at the ambient sound, but this time, we will use the sound we load in a slightly other or rather extended way. We will put the sound we loaded on the player model and hence pack it in the 3D scene. This will let the sound appear relative to the listener, wich may be for example the camera and will play the sound relative to it. For example the sound will only be hearable in your rear speakers if the node you parent the sound to is behind the camera.

To make audio ready to be placed in the 3D scene, we first need to set up an Audio3DManager which will be used instead of the global loader class. So add this lines in the main.py script.

```
…
from direct.showbase.Audio3DManager import Audio3DManager

        …
        self.hud = Hud()
        self.menuMusic = loader.loadMusic("assets/audio/menuMusic.ogg")
        self.fightMusic = loader.loadMusic("assets/audio/fightMusic.ogg")
        base.audio3d = Audio3DManager(base.sfxManagerList[0], camera)
```

Having that, we now also have a global audio3d manager that we will use for our 3D placeable audio.

We will also use that right away in our Player class to add some hit and step sounds to the characters.

```
    CollisionNode,
    KeyboardButton,
    AudioSound)

        …
        characterHitRay = CollisionSegment(0, -0.5, 1.0, 0, -0.8, 1.0)
        characterColNode.addSolid(characterHitRay)

        self.audioStep = base.audio3d.loadSfx("assets/audio/step.ogg")
        self.audioStep.setLoop(True)
        base.audio3d.attachSoundToObject(self.audioStep, self.character)

        self.audioHit = base.audio3d.loadSfx("assets/audio/hit.ogg")
        self.audioHit.setLoop(False)
        base.audio3d.attachSoundToObject(self.audioStep, self.character)

        …

    def enterWalk(self):
```

```
        self.character.loop("Walk")
        if self.audioStep.status() != AudioSound.PLAYING:
            self.audioStep.play()
    def exitWalk(self):
        self.character.stop()
        if self.audioStep.status() == AudioSound.PLAYING:
            self.audioStep.stop()

    def enterWalk_back(self):
        self.character.loop("Walk_back")
        if self.audioStep.status() != AudioSound.PLAYING:
            self.audioStep.play()
    def exitWalk_back(self):
        self.character.stop()
        if self.audioStep.status() == AudioSound.PLAYING:
            self.audioStep.stop()

    def enterHit(self):
        self.character.play("Hit")
        self.audioHit.play()
```

With that we have attached the sounds to the player model and will play them at the respective fitting states. Due to them being attached to the 3D audio manager and the 3D scene we will now hear the sound as seen from the camera from the players position.

What you also have to keep in mind is, if you remove the node from the scene, you also need to remove the audio attached to it as it otherwise will rise an error that it can't place the sound correct.

So add the following line to the stop method of the player class before you remove the character node.

```
        base.audio3d.detachSound(self.audioStep)
        base.audio3d.detachSound(self.audioHit)
```

For further information and more 3D audio specific functionality, take a look at the http://www.panda3d.org/manual/index.php/3D_Audio manual page.

That's all we will talk about for audio usage in Panda3D applications.

Particles

This chapter will teach you how to use Particles in your games. Particles can be used to add many various effects like fire, snow or visualize wind as well as performantely render many objects like the trees of a wood.

To use Particles in a Panda3D application, you need to create a particle file which describes the Particle effect. Panda3D comes with an application called Particle Panel, which is located in the particles sample folder that should be installed with the SDK. Make sure you have Tkinter and the Python megawidgets (PMW) installed. The Particle panel is a rather simple to use application and it will display the changes you make in it in a separate Panda3D window so you can use it interactively. The particle panel has quite a lot of features in it and as there are so many ways you can set up your particles we will just describe how to set up two particle effects we will use in the game. For a more full description of the particle panel, take a look at this Panda3D manual page https://www.panda3d.org/manual/index.php/Using_the_Particle_Panel.

Blood

The first example where we will use particles for is to visualize successful punches of the players. Therefore we will use a small image of a blood splat and load that into the particle panel, add a simple rotation and set a few other variables to make it look good for our purpose.

So after you have opened the particle panel, you should see a Panda3D window which has a bunch of floating white points in it. This is the main window, where you see the particle effect in action. The other window is the configuration panel, where we will set up our effect.

In the first section of the particle panel we will set up the system for the effect, this will set how many particles should be generated, how much they will spread from its origin the systems lifespan and the position at where the system is located. This will be the base settings that apply to all particles generated by this system. For our first system we will have a Pool size of 2 and a birth rate of 0.1. The rest of the system can stay as is. Next we get to the factory. The Factory will be responsible for creating particles and set how long they stay visible, give them a mass if physics should be applied to them as well as some velocity and some other things dependent on the factory type. The factory type will give the particles a few extra functionality like rotation about it's Z-Axis. In this section, we will set the factory type to ZSpinParticleFactory and set the following configurations. Life Span to 0.6, tick Enable Angular Velocity and set Angular Velocity to 155 and Angular Velocity Spread to 5. This will rotate the two particles around it's Z-Axis (the axis that points into/out of the screen). Next we move on to the Emitter section of the particle panel. For this particle effect, we will use a PointEmitter which will produce all the particles on one specific point. There are also a lot of other types of emitters basically the emitter type just describes the geometrical appearance where in or on the particles may get initially placed. For our effect, we will set the following values. Radiate Emission checked, Velocity Multiplier 0.5 Velocity Multiplier Spread 1, radiate origin 0, 1, 0.4. The rest can be left as default. The last section we will discuss for

this particle effect is the Renderer. This is used to specify the appearance of a single particle. For the effect we will achieve here, we will use the SpriteParticleRenderer which will render an image for each particle. To get our image in there as the sprite, we need to add a texture in the Texture tab and set its path correct which will be assets/fx/BloodSplat.png for our example. The alpha mode should be set to ALPHA_OUT to let the particles fade out over time. The next thing we will set for this particle effect is the scale, move to the scale tab in the Renderer section and set the Initial Scale values to 0.2 and tick the Anim Angle checkbox. Finally you need to set the Color Blend mode in the Blend tab to Madd to respect the images alpha bits.

Now you should have particles displaying the image you set slowly rotating and falling downward.

After creating the effect, we just need to load it into our game and use it at the correct time.

In the main.py template there already is everything set up to enable particles in the application, otherwise you'd have to enable them via base.enableParticles().

Now within the Player class in the getHit function add the following lines to enable the particle effect whenever the character got hit.

```
    def gotHit(self):
        if not self.canBeHit or self.isDefending: return
        self.bloodsplat = ParticleEffect()
        self.bloodsplat.loadConfig("assets/fx/BloodSplat.ptf")
        floater = self.character.attachNewNode("particleFloater")
        if self.character.getH() == 90:
            floater.setPos(-1, 0, 1)
        if self.character.getH() == -90:
            floater.setPos(1, 0, 1)
        self.bloodsplat.start(parent = floater, renderParent = render)
        taskMgr.doMethodLater(0.5, self.bloodsplat.cleanup,
            "stop Particle", extraArgs = [])
```

The first lines are there for instantiating and loading the particle effect. The floater will be used to place the effect respective to the character so it will be visible in front of it. After that we start the particle effect and start a task so the effect get stopped and removed half a second later.

Now we just need to add the following import statement somewhere at the top of the file.

```
from direct.particles.ParticleEffect import ParticleEffect
```

Having all that, you should now have a nice hit visualization on the characters.

Note that you can disable particles within the games config prc file by setting particles-enabled to false via #f or 0.

Falling leaves

The next Particle effect we will build is for our first level. We will let the particles render leafs that slowly fall to the ground directly in front of the camera. Most of the settings we do in the particle panel for this effect are quite similar to the ones we had before on the blood splat effect. We just differ with some of the values in the configurations we already know from before. The only new thing for this will be in the Emitter, where we will use a LineEmitter type this time. With that we just change the size and direction for the particles to start from and fly to. To see the values of the other variables, just take a look into the Leafs.ptf file by either open it up with a text editor or load it into the particle panel. Note, dependent on where you have the particle panel script located, you may need to add your source directory to Panda3Ds model path or change the textures relative path accordingly.

Now, as this particle effect will be used in our level, open up the arena.py script and add the following lines.

```
…
from direct.particles.ParticleEffect import ParticleEffect
…

        self.sunNp = render.attachNewNode(sun)
        self.sunNp.setPos(-10, 10, 30)
        self.sunNp.lookAt(0,0,0)

        self.ambientSound = None
        self.levelParticles = None
        if arenaNr == 1:
            self.ambientSound = loader.loadSfx("assets/audio/ambientLevel1.ogg")
            self.ambientSound.setLoop(True)

            self.levelParticles = ParticleEffect()
            self.levelParticles.loadConfig("assets/fx/Leafs.ptf")
            self.levelParticles.start(
                parent = render2d, renderParent = render2d)
        …

    def stop(self):
        self.arena.hide()
        render.clearLight()
        self.ambientSound.stop()
        render.clearFog()
        if self.levelParticles != None:
            self.levelParticles.cleanup()
```

This will enable the the particles for the first level and stops them as soon as the level get closed.

P3D shader system

Panda3D comes with a so called shader generator. The shader generator will enable some premade shaders for your game to greatly enhance the graphical appearance of it. By default as soon as the shader generator is activated and set the node that it got activated for or for all nodes in the 3D scene if set to render will be smoothly shaded with a per-pixel shader. Additionally to that it will also enable drop shadows which we will describe in the next section. Also it will enable the texture enhancement applied to the models like normal- gloss- and glow-mapping and a few other things.

The shader generator on the other hand is a quite extensive addition to your game. It will drop your games performance quite a bit, especially if it is used in a wrong way, so a general knowledge about shaders and how to use them would be beneficial.

To activate the shader generator you need to call the following method for all nodes that should be handled by it. The following line will enable it for all nodes in the 3D scene.

```
render.setShaderAuto()
```

For a more detailed description on limitations and usecases for the shader generator take a look at the http://www.panda3d.org/manual/index.php/The_Shader_Generator manual page.

Drop shadows

Having the Panda3D shader generator active, drop shadows will be added to the game when using spotlights. As we already added a spotlight to our arena which represent the sun in the arena, we just need to add a few things to it.

First we need to set the size of the shadow map in pixels by adding the following line in the arena.py scripts __init__ method somewhere around our already existing light setup.

```
sun.setShadowCaster(True, 2048, 2048)
```

Then again to the sun light node, we need to add a lens from which the shadow can be created, means everything seen by the lens will have a drop shadow except the objects are specifically set to not be accused by the light.

```
        …
        sunLens = PerspectiveLens()
        sunLens.setFilmSize(50)
        sun = DirectionalLight("sun")
        sun.setColor((1, 1, 1, 1))
        sun.setShadowCaster(True, 2048, 2048)
        sun.setScene(render)
        self.sunNp = render.attachNewNode(sun)
        self.sunNp.setPos(-10, -10, 30)
        self.sunNp.lookAt(0,0,0)
```

```
        self.ambientSound = None
        self.levelParticles = None
        if arenaNr == 1:
            sunLens.setNearFar(25,45)
            sun.setLens(sunLens)
            self.sunNp = render.attachNewNode(sun)
            self.sunNp.setPos(-10, -10, 30)
            self.sunNp.lookAt(0,0,0)

            self.ambientSound = loader.loadSfx(
                "assets/audio/ambientLevel1.ogg")
            self.ambientSound.setLoop(True)

            self.levelParticles = ParticleEffect()
            self.levelParticles.loadConfig("assets/fx/Leafs.ptf")
            self.levelParticles.start(
                parent = render2d, renderParent = render2d)
        elif arenaNr == 2:
            sunLens.setFov(120, 40)
            sunLens.setNearFar(2,10)
            sun.setLens(sunLens)
            self.sunNp = render.attachNewNode(sun)
            self.sunNp.setPos(0, 0, 5)
            self.sunNp.lookAt(0,0,0)
```

Having that, you should see some nice drop shadows in the arena.

Note that having many lights in a scene will now reduce the performance extremely due to the shadows and per-pixel lighting being activated, so make sure to keep an eye on the count of your lights and place them effectively.

Fog

Enabling fog is another very simple mechanic to add a nice touch to your game. It will prevent the player from seeing too far as well as give your levels a kind of depth. To enable a simple fog in your game, just add the following lines of code in the arena class.

```
            self.ambientSound = loader.loadSfx(
                "assets/audio/ambientLevel1.ogg")
            self.ambientSound.setLoop(True)

            self.fog = Fog("Outside Fog")
            self.fog.setColor(0.3,0.3,0.5)
            self.fog.setExpDensity(0.025)
        elif arenaNr == 2:
            sunLens.setFov(120, 40)
            …
            self.sunNp.lookAt(0,0,0)

            self.fog = Fog("Temple Fog")
            self.fog.setColor(0,0,0)
            self.fog.setExpDensity(0.065)
```

This will add a simple exponential fog that will move with the camera and have its density raised from the camera into infinity by the given setExpDensity value.

There also is another way to add fog which will not move with the camera and stay at the position where it was placed in the scene. For a more detailed description of it just take a look at this manual page http://www.panda3d.org/manual/index.php/Fog.

Crips Graphics

The following section gives you some tricks to get crisp non blured graphics for your game. Especially in the UI it is important to have crisp graphics to enhance the usability and readability of your application.

Antialiasing

Antialiasing is a method to smooth the edges of 3D models. Panda3D can automatically enable antialiasing for all models but also has specialized antialis modes that can be directly set on nodes that may not be handled correctly by the automatic antialias method.

To enable antialiasing for the whole scene you can add the following line somewhere in the Main class's init method.

```
render.setAntialias(AntialiasAttrib.MAuto)
```

and the following two configuration variables set either in the .prc file or via the loadPrcFileData method before you run the ShowBase instance.

```
framebuffer-multisample 1
multisamples 2
```

Texture Scale

You may notice that some textures, especially if used for the UI may look a bit blured. This is because Panda3D automatically scales textures to the next closest power of two size by default. This was important back then, when graphics cards where not capable of drawing non power of two textures, though as of today, most cards should be able to handle non power of two textures just fine.

The simplest way to tell Panda3D it should not automatically scale the textures is to set the following configuration variable to none.

```
textures-power-2 none
```

By default, this is set to down and hence downscales all non power of two textures.

You may also set the following variable to a true value if you want panda to check wheter the opened window supports this feature.

```
textures-auto-power-2 1
```

After that you should see much crisper textures in your application.

Font Scale

Aside of automatic texture scaling you may also set the pixel per unit value of Fonts, which will make them much easier to read. The most simple way to achieve this for all DirectGUI elements using the default font is to set it the following way.

```
DGG.getDefaultFont().setPixelsPerUnit(100)
```

This sets the number of pixels in the texture map that are used for each onscreen unit. Though note, that this requires more texture memory the higher the number you set.

Debugging and Testing

Cheats and the in game terminal

The first thing you may find useful for debugging is a in game terminal emulator which can be used for entering commands that the application knows how to handle or also for testing purpose some cheats like making the player invincible.

Panda3D doesn't have a premade in game terminal for you except a small on screen debug class which we will talk about later. So you could make an in game terminal by your own. Basically this could be just a onscreen text like the onscreentext class or directly a textNode for the output of information and a directGUI DirectEntry that lets you enter the commands. The DirectEntry will already have most of the functionality you need to enter text and sending it via the enter key, but you may also extend it with things like autocompletion and other nice to have things.

Direct tools for debugging and performance tuning

Panda3D comes with a few tools that will help you for debugging your game as well as finding performance bottlenecks.

On Screen Debug (OSD)

The first we will talk about is the on screen debug class. This is a class from the direct tools that will display a small HUD for debugging where you can add anything you like. This is especially useful for fullscreen debugging when you only have one monitor and can't see the terminal.

To use the On Screen Debug class you first need to import it with

```
from direct.showbase.OnScreenDebug import OnScreenDebug
```

If you then instantiate a variable with that class you can use it by calling the append clear and at the end render method to add, remove and show the OSD.

For an example of how to use the class you may take a look at the onscreendebug.py file within the core folder of the panda3d-code-collection.

PStats

The next thing we will show you is PStats. This is a tool delivered with the Panda3D SDK which can be used to find performance bottlenecks in your application. It is an application which has to be first started via the terminal either on the local machine or also usable over the network to make sure it doesn't draws to much performance from the PC. Additional to the PStats application the game has to be made ready via setting a configuration in its .prc configuration file or also the global

config.prc if you just have that. The configuration variable names you have to set to a positive boolean value is the following:

```
want-pstats           #t
```

Now if you open a terminal and start PStats you should see a window opening which states “Listening on port 5185”. If you then start your application, you will get another PStats window that will show you some diagrams that get updated over time with information about your application. These information can then be searched through to find what draws the most performance. To see where and what causes performance drops you might need a bit of training and need to check over and over again to build some knowledge on what can cause which behaviors but after some time, this application may be a big help on enhancing the performance of your game. You can also extended your game with additional things that get pushed to PStats wich will then add more details to it and show you what takes the most time in more detail.

For further description on what you can do with PStats and how to use it, take a look at the following manual page
http://www.panda3d.org/manual/index.php/Measuring_Performance_with_PStats

Unittests and Continuous Integration (CI)

The last thing you should add to your game are unittests. Unittests are used to test the functions you add to your game and check if they keep functioning even if you made some changes. Python already comes with a unittest framework which should work for most cases. Even tho it might be a bit hard to write tests for the 3D scenes, you sure can use unittests for things like checking the mathematical functions with values where you know the expected result. In combination with intervals, you may also be able to test 3D behavior for example if you move a character into a trigger collision and check if the event got thrown some variables that may be set on that collision got set correctly.

If you want to know more about unittests take a look at this python documentation:
https://docs.python.org/2/library/unittest.html

Now if you have added a few unittests, you may also implement a continuous integration (in short CI) for your game. A CI will get the most recent code from wherever it is stored then dependent on how it is configured, run all unittests given to it and finally build the application if applicable. If any of the things in the progress fails it will let you know by for example writing an email to you stating the errors so you can quickly find and fix them. This way you also quickly know if something broke since the last commit to a code repository. There are a few frameworks you can use as CI one for example is Builtbot and another one Jenkins. For more description on CI for python you can also take a look at this documentation http://docs.python-guide.org/en/latest/scenarios/ci/

Chapter 4: Release

Congratulations, you made it this far and should have a playable game by now.

This chapter will teach you how to make your game release ready by adding things like a manual, some icons and all those things you would expect from a nice game and pack all that up in an installer for any of Panda3Ds supported operating systems. Finally we will release it to the public through various stores and self publishing as well as advertise it and try to generate some revenue from it.

Manual and game release content

The first thing we do before we release our application is creating some release assets like the manual, a logo and some icons.

Manual

Let's begin with the games manual. It will describe how to install and deinstall the game as well as how to start it and then use it. Some manuals also contain some hints and tips and tricks for the players that really read the manual. Though, keep in mind that in the end your game should still be intuitive enough that it would also be possible to use it without reading the manual by either adding hints within the game or by having a gameplay tutorial within it which may be for example the first few missions in the game. Anyway, let's take a look at how we can write a manual. For this task, you can use a desktop publishing application just like scribus is. This application will let you create and design a manual quite easily. As the design of a manual is up to you and your designers, we just show up what must and can be in a games manual.

The first page should be a nice cover that attracts the users attention it should contain the applications name and maybe somewhere the note that this is the manual to it.

Following come the index which can be automatically generated by scribus or also text writing applications like libreoffice where the texts could then be imported into scribus.

After the index pages the main manual sections can start with the installation and deinstallation section. Those sections should have a step by step guide on how to get your application on and later on again off of the users machine. This section should be available for each supported operating system. This section may also contain a description on how to start the application after it got installed on the system.

Next to that you can either describe the controls with for example some images of gamepads and keyboards with descriptions of the various keys the application uses or directly jump in to the main menu and describe how to use the menus and what they are good for.

Having that done, the next chapters of the manual will describe the general gameplay and possibly add some hints even hidden secrets for the players like maps or where some special objects can be found in the game.

After those you can add some pages describing where the users can get support and all that legal stuff like licenses or the credits. Finally even though only useful for printed versions of the manual you can leave a few pages blank or with lines so that the user can write down some notes on them.

Now if you finished the writing of the manual, you should export it as a PDF document so it can be added to the release content and would also be print ready.

Logo

The logo is one of the most important artworks of your application. It will be the main image which users will remember when they see it and should instantly be able to tell to what it belongs. Hence a logo should be kept quite simple and pleasant to the eye with repetitive patterns within it. Also it must make sure to represent the general style of the game, for example if the game plays in a medieval setting with many sword fighting, you could use a sword and shield in the logo or if it's a SciFy space game use on of your most recognizable spaceship designs or something else that might reminds the users of this setting. You can also have a visual representation of the games name if it for example is the name of a thing. For example a soccer game named “kick that ball” could have a soccer ball and a players foot in its logo.

Now let's take a look at how you can design your logo. First of all, you should make a logo design session where you discuss with others how the logo could look. You should even have a few logos to decide from or take the best parts out of to create a new better one. Later if you have decided which logo design you want to take, you should create it as a vector graphic as those can be easily scaled to almost any size you need it for as well as being able to add or remove objects. For example if your logo consists of a text in front of an image you could remove the text and use the logo slightly grayed out in the background of manual pages or websites. You can even bring the the logo inside your game and display it somewhere in the world either pregnant or somewhere slightly hidden and unobtrusive. Finally, just use the logo wherever you want and try to slap it on anything that belongs to the game, so the user sure will recognize it after a while seeing it everywhere.

Icons

Icons are quite similar to the logo of the game in means of how they are made. After all, you could also use the logo as your applications icon. Though some games tend to use a different artwork for their icons which will be visible in the systems menus and on its desktop. Icons therefor should be very recognizable so the user quickly finds the application even in a rather filled up desktop or menu. An Icon should also just like the logo be created as a vector graphic image so it can easily be scaled to various sizes. Though if the icon gets to small, you should may have a separate vector graphic file as otherwise small details may get lost or the icon itself may look dirty and becomes unrecognizable. Icons should be available in as many sizes as possible, even though, the most common ones you usually find on all operating systems would be 16, 32, 48, 64, 128, 256 and 512 pixels square. You can also create larger ones but mostly you won't get lower than the 16px one as otherwise it really get hard to see what the icon tries to display. Now to the icons style, most of the time you will have a nice image which resemble the overall style of the game, but to make it even more recognizable, some applications have also used a slightly stylized font and just used the abbreviation or just first letter of the applications name.

The icons should be exported to PNG file format as well as being shipped in vector graphic format like SVG as some systems are able to use those too.

Licensing

As soon as you going to release a game or any other application, you have to think about the license you want to use for it to make clear to others how what they can use it for and possibly also share it with others. Licenses will also help you if you ever get into a legal dispute with someone who has used your application in a wrong way as he foremost must agree to the license the application is under.

As license I'd always recommend open source licenses as they would mostly benefit your application and the development of other technologies. Choosing which one of the open source licenses to take is a quite ethical question so I'll try to just point out a few things.

And note, open sourcing your application won't prevent you from making money if it's that what you're after, but we will point out on how to make money out of your application in a later section.

Now to the open source licenses, as mentioned, there are quite a few to choose from. There are those that decides to be copyleft and others that'll be copyright. In some countries there's even the public domain which will be the most open thing you can achieve. For a little explanation on the difference between copyleft and copyright, take a look at this website: https://www.gnu.org/gwm/libredocxml/x53.html

Starting with copyleft licenses, the most known one and also rather commonly found license is the GNU general public license. or in short GNU GPL. It already has a few different versions. It is provided and endorsed by GNU and the Free Software Foundation (in short FSF). There also is a kind of copyleft version of the CreativeCommons (in short CC) license namely the CC Share Alike which also is a quite commonly used license.

To get to some copyright type licenses there are also a bunch of good and well known licenses around that you can use, for example there are the BSD licenses, Apache license and most of the CreativeCommons licenses. Those licenses contain the copyright holder and ensure the author will be stated with each copy of the source.

Aside of the copyright and copyleft licenses, you also must make sure the license is valid for your country as not all licenses may be accepted in every land.

You can also use multiple licenses on your application like one for other open source users and one for commercial usage as well as split up the licenses so that the applications source code has one license and its assets like music, textures and models have another license. This tutorial for example uses two licenses as stated in the copyright section at the beginning of this tutorial.

For a rather long list of available open source licenses sorted in categories take a look at the following website http://opensource.org/licenses/category and, to help you with choosing a license here are a few websites that will guide you by asking simple questions and listings of licenses:

http://choosealicense.com/ and http://www.gnu.org/licenses/license-recommendations.html

Pack the game as executable and installer

Now you should have all the assets for your game and be ready to pack anything up into an executable and installer.

Panda3D in it's stable releases ships with a few nice tools which makes this step really simple and straight forward. The first thing we need to do is packing the game's assets and source code into Panda3Ds own executable and kinda operating system independent file format named p3d. This file will then be able to be played via the Panda3D Runtime which we can either automatically install with the installer that we can create using the Panda3D deployment tools or put on a website to let the game be playable in a web browser.

Before we start the packing, we need to prepare a few things. First to have a clean structure, where we can pack the application, take the source and asset folder and copy them within a new directory called DEPLOYMENT and remove all file that should not be contained in the end product like all binary python files like .pyo or pyc as well as backup files and the like.

So you should have a structure like this

/DEPLOYMENT/
/DEPLOYMENT/License.txt
/DEPLOYMENT/[all icons for the game]
/DEPLOYMENT/game/
/DEPLOYMENT/game/[all code files and other necessary files]
/DEPLOYMENT/game/assets/
/DEPLOYMENT/game/assets/[all assets and subfolders]

Additional to those files you should also create a config.prc file within the DEPLOYMENT directory which then will be used automatically to set some default values at startup time of the application. This could for example be something like setting the fullscreen mode or audio volume. Basically anything which is described in the following Panda3D manual page: https://www.panda3d.org/manual/index.php/List_of_All_Config_Variables

For our game you should at least add a window-title Tatakai no ikimono into this config.prc file.

Next we need to create a certificate that we will use to sign our game. Even though, this is mostly only useful for games that will be put on a website it definitely adds to the professional look and feel of the game itself if it is signed.

Now, unless you have a valid and by a official authority like StartCom (StartSSL), GoDaddy or Symantec trusted certificate you need to create a self signed certificate. To create a self signed certificate, you can for example use the OpenSSL tools which comes preinstalled with most Linux distributions. The openssl commands can then be called as follow

first create your private key which we will call main.key. This name can also be anything else like your companies name or something that you recognize for what it's used for.

Note, the second command will ask you for some additional entries like your companies name, real name or FQDN, email address and the like. Not all of these have to be entered but the more details you give, the trustworthier your certificate will become.

```
openssl genrsa 2048 > main.pem
openssl req -new -x509 -nodes -sha1 -days 365 -key main.pem >> main.pem
```

Having the certificate, you now have everything ready to pack up your application with Panda3Ds deployment tools which we will describe how to use now.

Within the DEPLOYMENT directory call the packp3d command as follow

```
packp3d -d game/ -x Cursor.ico -n x11 -r openal -S main.pem -o tatakai-no-
ikimono.p3d
```

This will generate a file containing everything from within the game directory and will be called p3d-tutorial.p3d. It can be run by using the panda3d command which comes with the Panda3D runtime. After all, the packp3d command is also just a panda3d application and hence needs the Panda3D runtime to be installed. Using the -x option tells the application which files should be extracted when the application is run, tough note, this will only be respected if the application is also packed up with the pdeploy tool which we will talk about a little later. Next we also need the -n option which will add files that won't be added by default as incompressible files. There is also a -e option which can be used to include extra files in compressed mode. Additional to those options we also have the -r option. This one will give us the possibility to add extra packages with functionality to our game which otherwise would be left out to have a most stripped down version as possible. For our game, we just need OpenAL for playing audio files and hence add it with the option. To see what else is available for the -r option, just take a look at the following manual page https://www.panda3d.org/manual/index.php/Referencing_packages

Finally there is the -S option which we give our certificate that we use to sign this application.

Note, by packing your game with packp3d there are a few things you need to know. First of all the maybe most important thing, all your .egg model files are automatically converted to .bam binary files, so make sure to not have .egg extensions in your model loading paths. Also note, each .p3d file is basically just a multifile, which is Panda3Ds container format. Because of that, some files must be extracted first before they can be used like we mentioned before with the .ico and .x11 cursor files.

If you run this command and if it run through successfully, you can already start your game by calling the following command in the DEPLOYMENT directory

```
panda3d tatakai-no-ikimono.p3d
```

This should open up a window which may first show you a loading screen telling you it's downloading and installing the engine. Afterwards it should directly start into the game.

This file can already be packed inside a website and be played within a web browser. For further details on how to get it on your website just take a look at this and the following Panda3D manual pages: https://www.panda3d.org/manual/index.php/Distributing_via_the_web. Note, you can also use pdeploy to generate a basic website for you that embeds the just created p3d file.

Now we will take a closer look at the pdeploy command to generate an installer for our game for every platform Panda3D supports. There are quite a lot of information we will hand over to the command, so here is the command line that can been used to pack the game. Note before you run this command, you need to create the release directory within the DEPLOYMENT directory we will use in the command.

```
pdeploy -N "Tatakai no ikimono" -v 15.12 -o release/ -s -l "2-Clause BSD
License and CC-BY-SA" -L License.txt -a org.grimfang-studio -A "Grimfang-
Studio" -e "info@grimfang-studio.org" -i icon16.png -i icon32.png -i
icon48.png -i icon64.png -i icon128.png -i icon256.png -i icon512.png
tatakai-no-ikimono.p3d installer
```

Let's go through it and see what's in it.

First the -N option. This simply represents the full visible name of the application.

Next comes the -v which is used for the version number that may be seen in the installer.

The -o option tells the command where the generated packages should be stored in. In our example we will put them into an extra directory called release.

Now comes a small but important option, the -s. This will let the installers be fully self containing. Means the end user does not need a Internet connection to install the games. If this switch is omitted, the installer will be generated a bit faster, but the user needs a Internet connection to be able to get the Panda3D libs installed while the installation. This way the installation on the users side also will take longer.

Next to the -s switch comes two options for giving the license information of the game.

The -a and -A switches are used to set the games maker company followed by -e for a support email address that the users may use to contact if there are any problems with the game.

And last but not least, we add the icons we have created for the game.

After we have set all the options for the command, we just need to give it the previously created .p3d file and tell it what the command should create. In our case this is an installer, but it can also be a html page or standalone executable.

Note, there also is a -P switch that can be used to only build for some specific platforms like linux_amd64, linux_i386, win_i386. Therefore you just hand over a a coma separated list to it.

For more information on the parameters just call pdeploy -h

To not need to type this command every time, you should store it in a script like a shell or batch file.

Share your game

From here on, you should have a full and nicely packed up game that's ready to be spread to the people.

You can now try to reach some video game publishers and try to get them sell your game or publish it by your own. Unless you have a really good AAA grade game it may not be that likely to get your game published by the big players in the business, so we will take a closer look at self publishing.

There are quite a lot of places on the web where you can start selling and sharing your application. One of the maybe most known is steam greenlight, which makes it possible to get your game in the steam store. Though keep in mind, not every application posted there will make it into steam, there still is a bit of luck and a good userbase that will vote for your game. Other more simple ways to get your game to the users are the Ubuntu Software Center (USC) or a personal amazon shop. On those you don't have to hope on getting enough votes to get in there. For the USC you just need to pass the review process by canonical to get your application in. Aside of those, there are many other services like itch.io and gog.com where you can try and get your game into to sell it there.

Of course you can also use your own website, if you have one, to share your game with others. If your game is an free and open source game, you can also use services like launchpad.net or sourceforge.net to share the code and binary files of your game.

Advertise your game

As your game is ready to be downloaded and bought by the users, we also need to make sure people will hear about it and know your game even exists as you can't expect it to become famous without telling someone about it.

There are many ways to advertise your game and get the masses to recognize it. The most simple and also a very effective way is by self advertising the game. By self advertising you do everything by your own, reach out to people, talk about your game and make it popular. This is also a very time consuming method but if you really want to get your game famous, definitely a way to go.

Other than self advertising, you can also get you some agents that do the advertising for you on various platforms. If you even have the luck to get a publisher that will publish your game, they will keep track on the advertisement by themselves. Though, this ways even though may not need as much time as self advertising, they sure will need you to get more money in your hands to go these ways.

No matter if you go fully on your own or a mixture of self advertising and getting others that will help you, here are a few ways how you can advertise your game.

The first one is by word-of-mouth advertising. Just reach out to your friends, family and everyone you know to let them know about your game and tell them they should also spread the news.

Similar to this, you can also use the social media platforms like identi.ca, facebook, twitter and all the like to get your game known. If you're lucky you may even get a let's player to play your game and share it on youtube, vimeo or any other video sharing platform you like. Of course you can also create some nice videos of your game like let's plays as well as trailers by yourself and put it on those video sharing sites.

You can also put some news and advertisement for it on your own website if you have one. If not, you can get free or cheap webspace all over the Internet and create some nice website with it. You can also try to get your banners on other websites that are dedicated to video games or the user group you try to reach.

If you really want to go big, you can also try to get your game advertised with posters or even if you can spend a big amount of money do TV commercials. Depending on where you live, you need to contact local poster or TV channel providers to get your ads out there. Aside of that which may also be a bit cheaper, is getting your game advertised in games magazines.

Now, no matter where you going to advertise, make sure to integrate Corporate Identity (CI) into your ads and games so people will recognizes games from the same company/organizations/teams. CI means that you have a specific set of recognizable styles which includes but is not limited to logos, fonts, color schemes and even pixel perfect definitions for styles of images and websites. To make sure everyone knows your CI you should have a document with all CI definitions in it which is publicly available.

Making money

Finally, the last section for the deployment of your application will tell you a few ways and strategies on how to get revenue from your released game.

All of this ways can be used no matter if your application is closed or open source, though for a description on why you should prefer open source, take a look at the later section “the benefit of open source” in chapter 5.

Following are some possible points on how you can create revenue from your applications.

1. Sell precompiled copies of your game

This was and still is the most common way to get revenue from your application. Sell the copies of your game as mentioned before either in online shops or selling hardcopies.

2. Periodic fee

Another way similar to selling precompiled copies is by charging a periodic fee every month/year

for letting the users play the game. This is for example in the mmo game genres very common way.

3. Ingame purchases

Selling items and little gimics within a game is getting more and more popular especially in mmo games. This way you have to provide extra content and assets for your game, that users can buy from a shop within the game or on the games website to get special items and features like extra clothes for the character to make it more individual.

4. Sell additions to the game

You can sell additional content to the game like mods, add-ons or level packs and the like. Just make sure the initial game is also fully featured and playable otherwise the user might turn down your game before he even gets to the point where he want to buy a new addition.

That's it, now you should know all the basics you need to know to create a full featured game and release it to the public.

Chapter 5: The next version

Now that we have a game released we can lay back a bit and enjoy the responses to it. But as to every game, there is so much more that can be added to it so many more features waiting to be implemented, so make yourself ready for the next version of the game.

The benefit of open source

One way to get new features in a released game is with the help of developers around the world that like your game. If you open source your game you make it possible to others adding features and things they like. This way your game should grow much faster than if it's just you or even your group developing it as every developer enhances the speed at which your application grows.

To make your game available to the public as open source, you can host the code and assets on a vast amount of hosters. Just to name a few well known code hosters, there is Launchpad, Github, Bitbucket, GNU Savannah, Google code and SourceForge. For more of them and a comparison of them, take a look at this wikipedia page:
https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities

You should also make sure to take a look at how other open source applications are run and which structures they use, the bigger your application get and the bigger the community is around your project, the more time you may need to put into leadership and management if you don't hand this over others in your community.

Additional features

Lets take a look at what else could be added to our little tutorial game. We'll describe a few points here and give you some hints at how to implement them, but we won't give you the necessary code as this would exceed this tutorials size. But you should be able to get it together with the hints and knowledge we give you in the several sections following.

Gamepad support

One of the first additions you could add to the game is Gamepad support.

even though Panda3D currently doesn't natively support Game- and Joypads, you can use other open source libraries to add support for them. One of the more simple ways and also rather good supported way in Panda3D is using the PyGame library for gamepad support. For sample of how to use gamepads with PyGame within a Panda3D application, you can take a look at the panda3d-code-collection specifically the input/joystick_pygame.py script.

To let the user control the game via the joystick, you need to enhance the player script as well as add some more functionality to the menus so they become controllable using a gamepad if you won't let the gamepad directly control the mouse.

Extended character

The current character controls are quite simple and there are much things that can be added to a beat'em'up game. Following we'll list a few things that you can add by yourself and give you some hints on how they can be implemented in the tutorial code.

First of all you could implement some more attack animations and follow up attacks. For that you just need more animations for your 3D models. Now if you want to add them in the game you simply need to extend the FSM states. This is by far the most simple thing you can add. You can also use a few different versions of an attack and play them randomly when the player initiates the specific attack.

Another thing one also mostly sees in beat'em'up games it jumping and other physical stuff like getting pushed back by a hit or while airborne. As mentioned in this Tutorial at the collision section, there are a few physic libraries implemented in Panda3D but you can also use your own implementation for example by using tasks and calculate the new position at every frame. Then you just need to apply the force specific to the action that happens in the game. For example add an upward pointing force to let the player jump up for a given time and have a gravitational force which always draws the player towards the ground. Note that you need to enhance the arenas by adding a floor collision solid so the players wont fall through the ground.

Making the game more like real live with physically interactions, you may also want to enhance the collision detection. As already mentioned, you can attach collision solids to specific bodyparts of the character, which then may also be moved by the animation of the visible model if parented correctly. You should though still have a main collision model that will then solely be used to avoid moving though walls and the ground as it will definitely be more performant to use one collision solid for that. You can apply so called bitmasks to collision solids to define which collision solids can collide with other solids.

The next thing to bring more realism and more depth to the gameplay is by letting the character not only move on the X and possibly Z (jumping) axis but also on the y axis, so players will be able to dodge other characters attacks. To get this kind of movement, you need to dedicated a new keyboard key to it and enhance the movement function of the Player class. Also the camera update task function needs an update as it was just intended to move on the X-Axis. Aside of the camera also the players need to have an addition in the move or other task method to always look at the player except when an attack is initiated so other players really can dodge the attacks.

Most beat'em'up games also have combo attacks. Those might be quite simple to implement by adding a list where you store the successful attacks of the player and clear it whenever an attack misses or a timer runs out. For the timer you can use the dt time that we already used in the task methods. You may also vary with the attacks strength the higher the combo count is.

Finally you can add cinematic animations by adding intervals. Intervals are a feature of Panda3D that will be used quite often. It can for example be used to add moving platforms or doors but also, more specific to this kind of game, cinematic finishing moves by playing a few animations chained together.

Enhanced UI

The GUI is another part that can simply be extended by a bunch of commonly in games found things.

The first which may comes to mind is an options menu where you can change configurations of the game which may contain audio volume and muting options as well as graphic quality settings. If you by now also have added a singleplayer feature wher AI is used for the enemy characters, you may also set the difficulty in here. Also an important thing for most games is changing the keyboard and gamepad mapping, means the player gets the freedome to change the keys actions to his liking.

Having that many menus now you can also add a few graphical enhancements to it by adding some menu change animations like in and out fades. Those can also be rather simply be added with Intervals which have been mentioned before.

Another very common thing that all commercial games have is a intro and or logo movie and afterwards a title screen. The logo and intro movie can be a prerendered video file that will then be mapped on a plane and is just played at the start of the game. For a detailed description on how to

play back videos in Panda3D take a look at the following manual page.
http://www.panda3d.org/manual/index.php/Playing_MPG_and_AVI_files

Singleplayer

As we haven't featured the addition of a singleplayer mode in the game, we leave this up to you to implement. For a singleplayer mode you need to add and change a few things.

First of all you need to implement the artificial intelligence or in sort, AI for the game. The AI will take control over the enemy player and move it around as well as attacking you and get into defense mode to block your attacks. There are a few ways of adding AI to the game if you have freely moving characters you could add Panda3Ds implementation of some AI behaviors via the PandAI (see this section of the manual http://www.panda3d.org/manual/index.php/Artificial_Intelligence_%28PANDAI%29) Though, for our beat'em'up this may not be the most suitable set of AI behaviors, so you need to add a solution by our own or search for how other games in this genre have done it.

Next if we have the AI for the enemy, we need to enhance the GUI to be able to start either a single or multiplayer game. This can be done by either add a new button to the main menu and change the start button to a multiplayer button or change the character selection by let the second player be set as computer controlled.

Story mode

A story mode will add a bit of replayability and is especially nice for singleplayers even though the story might also be playable with more players at the same time.

The levels in a story mode may either be arenas but can also be pipe like levels where the player gets guided through. For those kind of levels you may need to enhance the camera control by adding fixed position in the levels or move the camera with the player. In those levels you may also add multiple enemies that the player/s have to fight against.

Between each level you then can add small clips or images which will move the story forward and describe whats happening in it.

Additional to the Levels you may also enhance the main menu just like described in the singleplayer section so that you have a button for the single player mode one for the multiplayer and one for the story mode.

Having a story mode, you might also find a save function useful for the player so they can save their position in the story to pause and later continue from that position.

Unlockable elements and achievements

Most games have unlockable elements and achievements. Therefor you first have to think about what kind of achievements and unlockable objects you want to have in it and how they can be unlocked. Those can for example be extra and hidden characters and levels as well as bonus content like a gallery or little achievements like medals or flags that the player can collect by doing something special, for example make a combo of 25 hits or get hit 1000 times throughout the complete game.

The unlocked things then also need to be stored in a kind of player profile so you may also think about making the player be able to create a few profiles and not store it in a “global” profile even though it would be possible to handle it all in a single profile which can be reset by the player.

For these things you should create a few new classes to store all the informations and put the unlocking in the main class as well as some additional events that may be useful in the other classes we used throughout the game.

Multiplayer over network

If you want the game to be playable by two persons that must not be on the same PC, you have to implement some network functionality. This will give the players the possibility to play over the Internet or LAN against each other.

There are a few ways to add Multiplayer Networking support to a Panda3D application. Panda3D for example has implemented a kind of low level networking feature as well as a rather high level but not that good documented system called distributed objects.

There also are community implementations of various networking systems. In the end it all depends on what your specific needs are. For this tutorial for example a self written approach using the low level networking system of Panda3D may be the most simple and fastest way on getting Networking support in the game and, on the other hand, this way you'll also learn much about networking in general which will definitely be useful if you use one of the higher level systems as networking in general is a big and complex part which would definitely be to much for this tutorial.

Adding easter eggs and hidden secrets

The last thing and also famous in big games or general software applications are the so called easter eggs. Those are little jokes the developers put in a game that the players might occasionally stumble upon. Those could be many kind of things. For example change your characters look to something else like a famous movie star or put him in a funny costume. Also things like changing the music or adding some hidden areas are very common in games.