Game tool development tricks, tips, and concepts from industry experts

ADVANCED

Game Development Tool Essentials

Alessandro Ardolino, Remi Arnaud, Paula Berinstein, Simon Franco, Adrien Herubel, John McCutchan, Nicusor Nedelcu, Benjamin Nitschke, Fabrice Robinet, Christian Ronchi, Gustavo Samour, Rita Turkowski and Robert Walter

Apress®

Contents at a Glance

Introduction

The computer game industry isn't what it used to be. Early on, which wasn't all that long ago, developers focused on bringing the magic of arcade games to microcomputers, which was fun, but suffered from a computing environment that was technically and artistically limiting. However, as computing power exploded, so did developers' technical options and creativity, culminating in the sophisticated AAA titles that became so popular in the aughts. These marvels required large development teams, with complex and proprietary platforms that themselves required dedicated teams of programmers, and game development grew up; boy, did it.

In the last few years there has been a massive explosion in the growth of mobile and casual gaming, which has dramatically changed the nature of game development. Many successful products are now developed by small teams that do not have the resources to build the kind of complex tool chains AAA teams use. These developers cannot afford the luxury of specializing in one small part of a complex system. To build a modern game, typically in a web or mobile environment, you must be familiar with a wide range of technologies and techniques, and you must be able to turn your hand to meet the immediate need, which may be just about anything: one day asset management, the next capturing usage statistics, the day after passing conformance tests.

This book was written with the needs of the new developer in mind. We offer strategies for solving a variety of technical problems, both well-known and unusual ones, that our experts have encountered. There's quite a lot about COLLADA, as well as techniques for using the Web and the cloud in your pipeline, rapid prototyping, managing your files and assets, and optimizing your GUIs. We think there's something for everyone here and hope you agree.

Code samples are written in Java, MAXScript, Objective-C, Python, HTML, JavaScript, JSON, C, C++, C#, AngelScript, Xtext, and domain-specific languages.

We've divided the book into four parts:

- Asset and Data Management
- Geometry and Models
- Web Tools
- Programming

Asset and Data Management covers the critical issue of managing your assets in the game development pipeline. Two different aspects are described; both will help developers reduce their workload in the always daunting process of not only organizing original assets, but also tracking changes and versions. For example, Chris Ronchi's "Where Is It" chapter explains why it's important to follow a consistent and thought-through naming strategy and demonstrates how to create one. This basic but useful chapter attacks an area that involves no programming and no expenditure, but can help you save time and money just by using a basic "convention over configuration" approach.

The second section, Geometry and Models, focuses heavily on the COLLADA document format, describing how it can be used to bridge the gap between proprietary high-end tools and the requirements of small developers.

The Web Tools section offers hints on moving game development tools to the cloud as well as some particularly interesting ways in which readily available open source web development tools may be used. By adopting software like Django, for example, it's possible to build a comprehensive web-based gameplay monitoring and tracking system.

Finally, the Programming section offers help for developers who want to create their own flexible workflows. The emphasis here is on employing programming techniques that were originally developed to solve more general problems, such as the Command Pattern and the use of domain-specific languages (DSLs), to simplify the programming task. Each programming chapter describes not only the use, but the concepts behind the particular technique, so you can identify a variety of use cases and build up your armory of skills.

Just a quick word about the development of this book. Unlike almost every game development volume out there, this one was originally published independently by a group of experimenters—people with something to say who came together to try something new; it was titled *Game Tool Gems*. We self-published the book in both hard copy and ebook formats and sold it through Amazon. But it was a bit bare bones: no index, tech reviewed only by each other, laid out by editor Paula B. rather than by a fancy designer. But people liked it, including Apress's editors, and that's how this book with its new title, *Game Development Tool Essentials*, was born. Apress took the basic material, tech reviewed it six ways from Sunday, added that missing index, and expanded and updated all the chapters. And for that, we are most grateful.

We feel that it's critical to share information about the tools we use to create games. The better our knowledge, the faster and more efficiently we can work, and the more cool things we can do. That's why we wrote this book pooling the knowledge and experience of working developers. That fragile pipeline has plagued us long enough. Let's show it who's boss.

PART 1

Asset and Data Management

CHAPTER 1

Plug-in–based Asset Compiler Architecture

Nicuşor Nedelcu

From the beginning of the game creation process to the end, developers have two things to worry about: code and data (game art and other type of assets). In the past, data was formatted specifically for the one platform the game was about to run on. Now we have to format the same data for many different platforms. In order to satisfy this new requirement, we need access to source assets that can be compiled into a variety of targets. We also have more work to do, since special care has to be taken for each format.

However, there are ways to reduce the pain involved in this more complex pipeline. To make this process as streamlined as possible, I propose a plug-in–based asset compiler that can load converter plug-ins for the given asset types. The plug-in–based nature of the compiler can also help developers create their own plug-ins for any other special asset types they need. In this chapter, I describe how to set up and code such a compiler using an example of a texture converter/compiler plug-in. The platform you are going to use is Windows and the language is C++; with few modifications regarding the OS specifics, the code should work on other environments and can even be adapted to other languages.

Design

The underlying plug-in loading system can be a "traditional" dynamic-link library (DLL[1]) loading and querying for the proper interfaces. You will be using the Windows DLL API, but the code is almost the same for other operating systems. The DLL can export a special function that will return an instance of the converter interface (see Listing 1-1). The same goes for other platforms (OS/X, Linux), using their specific dynamic link library API implementations.

Listing 1-1. Creating the Asset Converter Instance Using the Exported DLL Function

```
DLL_EXPORT AssetConverter* createAssetConverter();
```

The interface of the asset converter looks like Listing 1-2.

[1]Wikipedia. "Dynamic Link Library." http://en.wikipedia.org/wiki/Dynamic-link_library.

Listing 1-2. The Asset Converter Interface

```
class AssetConverter
{
public:
  enum EType
  {
    eType_Unknown = 0,
    eType_Compiler = 0>>1,
    eType_Importer = 1>>1,
    eType_ImporterCompiler
                   = (eType_Compiler | eType_Importer)
  };

  AssetConverter(){}
  virtual ~AssetConverter(){};
  virtual bool convert(const char* pSrcFilename, const char* pDestPath, const Args& rArgs) = 0; //
Args class is a command line argument parser, not shown here. Basically holds a list of arguments
and their values
  virtual const char* supportedExtensions() const = 0;
  virtual EType type() const = 0;
};
```

The asset converter has a type that represents what the converter does with the given input file: compiles or converts. You make this distinction between compilers and converters because you would like to use compilers to compile data from your intermediate format to the final platform-optimized format, and converters to convert from third party formats to your intermediate format. An example of a compiler is `cube.json` (the intermediate format) to `cube.mesh` (final optimized format); of a converter, `cube.fbx` to `cube.json`.

You can also have a compiler and a converter in one implementation (`flag eType_ImporterCompiler`) that can handle third party and intermediate formats (for example, a `TextureConverter` that converts third party JPG/PNGs and compiles to a custom format like `.TEX`).

The convert method is the one called by the asset compiler executable when the given command-line arguments are passed to it, and they match the file extensions returned by the `supportedExtensions()` method. This function should return something like a file mask such as `*.jpg`, `*.tga`, `*.png`, or `*.texture`, so even a simple substring matching test can select the right converter(s). The command line arguments are shared for all the converters; each one can pick up its own arguments and their values.

By convention, the converters will be called first on the given assets, and after that you will call the compilers. Since you (probably) generated/converted assets from the previous step, now you can compile those intermediate formats into final binary optimized ones for specific platforms.

The main asset compiler executable will load all plug-in DLLs from either a specific folder or the same folder as the executable. You can use any kind of plug-in loading scheme. For example, you can have those DLLs with their extensions named .plugin, .converter, etc. In this way, you dynamically load only the eligible ones, skipping the unsupported/unknown DLLs.

Once a plug-in is loaded, you retrieve the address of the DLL exported `createAssetConverter()` function and instantiate the converter. Then, with all plug-ins loaded, you match each input asset filename with the return string of the `supportedExtensions()` of each converter. If the match is true, then you call the converter to take care of that file type. After that, you can continue to pass the filename to be handled by other matching converters, or you could come up with a stop Boolean return value so the file will be handled only once by a single converter and not by further matching converters if the return value is false. Even further, you could have some sort of dependency tree dictating when converters would be called after others have finished converting assets.

Obviously, another thing that speeds up the compilation/conversion process is multithreading.[2] In a first phase, you can schedule groups of files to be converted on separate threads. Then, when you convert a few files, the converters could spawn several threads to take care of a single asset. You must be sure, however, that the available cores are used/spread evenly, whether on a CPU or GPU.

Multithreading asset compilation can be a little tricky when dependencies are involved, so for this process to be safe, and to avoid problems arising from two threads concurrently modifying the same resource, you should build a dependency tree and put each main branch and its sub-branches and/or leaves on their own thread. Various methods for thread synchronization can be used, like mutexes and semaphores, each operating system having its own API for that. The main compiler class would look like Listing 1-3.

Listing 1-3. The Asset Compiler Class

```
class AssetCompiler
{
public:
  AssetCompiler();
  virtual ~AssetCompiler();

  bool compile(const Args& rArgs);
  void compileFolder(
    AssetConverter::EType aConverterType,
    const char* pMask,
    const char* pExcludeMask,
    const char* pCompileFolder,
    const char* pDestFolder);
protected:
  vector<AssetCompilerWorker> m_workerThreads;
      .......................
};
```

The main asset compiler class has the `compile(...)` method (synchronous call; it will wait until every asset compile thread finishes), which will take the actual command-line arguments. The `compileFolder(...)` method (asynchronous call; it will just start the threads) will process a given folder for a specific converter type, with a filename mask, an excluding mask, the actual compile folder, and destination folder for the output files. The class also has some worker threads for multithreaded processing of the input assets.

Example

The code in Listing 1-4 shows an example—a texture converter/compiler plug-in.

Listing 1-4. The Texture Converter and Compiler

```
class TextureConverter : public AssetConverter
{
public:
  TextureConverter();
  ~TextureConverter();
```

[2]"Multithreading." http://en.wikipedia.org/wiki/Multithreading_(computer_architecture).

```
  bool convert(const char* pSrcFilename, const char* pDestPath, const Args& rArgs);
  const char* supportedExtensions() const
{
     return "*.jpg *.png *.psd *.tex";
}
  EType type() const
{
   return eType_ImporterCompiler;
}
};
```

As you can see, the texture converter plug-in class will return all supported file extensions and their types, so the main compiler class will select it when appropriate.

Inside the convert method, the code will check the input filename and dispatch the logic to the specific image format handler.

This class can reside in a DLL, and you can have a single converter per DLL, but you can also have as many converter classes in a DLL as you want. In that case, the query function will just have to change to support multiple classes. See Listing 1-5.

Listing 1-5. A Plug-in with Multiple Converter Classes Inside a Single DLL

```
// this class must be implemented by the plug-ins
class AssetCompilerPlugin
{
virtual int getClassCount() = 0;
virtual AssetConverter* newClassInstance(int aIndex) = 0;
}
DLL_EXPORT AssetCompilerPlugin* createPluginInstance();
```

The exported createPluginInstance() will create the plug-in's class instance, which will take care of instantiating converter classes.

Other converter plug-in examples include an FBX converter, mesh compiler, prefab compiler, shader compiler, MP3/OGG/WAV converter, level compiler, etc. The plug-in system can be developed further with class descriptors, so you can have information about the converter classes without having to instantiate them unless they are needed.

Conclusion

Making the asset compiler modularized can yield huge benefits: shorter development time, the ability to extend and debug the tool, and happy third party developers who will use the tools since they can implement new converters/compilers for their custom data file formats.

Keep in mind optimizations like multithreading; dependency trees; CUDA/GPGPU operations to speed things; a CRC-based last-modified file info database so you can skip assets that haven't changed; and even safely running many compiler executables on the same data folders.

The solution can be implemented in various ways. The converter ecosystem can be detailed as needed so it will fit perfectly into the game engine's pipeline.

CHAPTER 2

GFX Asset Data Management

Christian Ronchi

Working in a software house is primarily about collaborating with other people, so the first thing to do when you start a new project is set up a pipeline that facilitates the flow of assets and information. Ignoring this important preparation can create confusion and waste time during production, so you want to make sure you do it right.

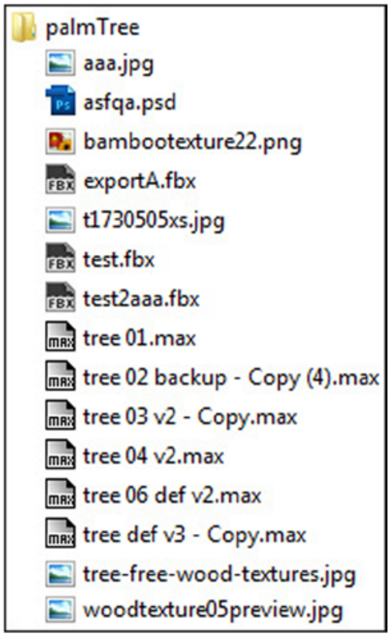
One of the most important considerations in setting up such a pipeline is keeping track of your assets. You don't want programmers and artists making changes to the wrong version, or losing the best version, or not being able to find that great character variation to show the director who needs to see it right now. Fortunately, it's not difficult to create a system that will keep these kinds of disasters from happening. What you need is

- One place for everything. Assets and project information should be stored centrally to keep them consistent. You might want to use a wiki or set up a common space (sometimes we use Microsoft SharePoint in our studio) where information can be constantly updated and available.
- Easy-to-understand file names and organization. Asset naming conventions, folder structures, and file organization must be simple, efficient, and intuitive.

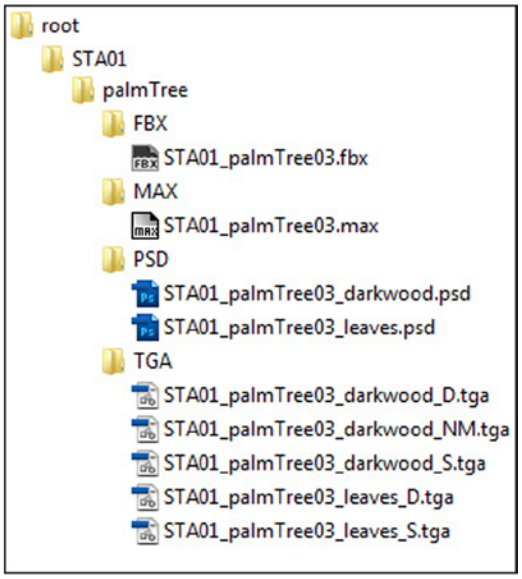
This chapter focuses on the second element of that system: file organization and naming conventions.

Folder Structure

Folder structures, file names, and their internal organization must be designed to be clearly interpretable by any person who needs to work with the project's assets. Figure 2-1 shows an example of bad organization of the directory structure/files applied to a common type of Autodesk 3ds Max project. Next to it, in Figure 2-2, you can see the same project with a simple, well-organized structure. In this example, we're using a train station with a palm tree.



```
palmTree
    aaa.jpg
    asfqa.psd
    bambootexture22.png
    exportA.fbx
    t1730505xs.jpg
    test.fbx
    test2aaa.fbx
    tree 01.max
    tree 02 backup - Copy (4).max
    tree 03 v2 - Copy.max
    tree 04 v2.max
    tree 06 def v2.max
    tree def v3 - Copy.max
    tree-free-wood-textures.jpg
    woodtexture05preview.jpg
```



```
root
    STA01
        palmTree
            FBX
                STA01_palmTree03.fbx
            MAX
                STA01_palmTree03.max
            PSD
                STA01_palmTree03_darkwood.psd
                STA01_palmTree03_leaves.psd
            TGA
                STA01_palmTree03_darkwood_D.tga
                STA01_palmTree03_darkwood_NM.tga
                STA01_palmTree03_darkwood_S.tga
                STA01_palmTree03_leaves_D.tga
                STA01_palmTree03_leaves_S.tga
```

Figure 2-1. *(left). A badly organized structure*

Figure 2-2. *(right). A clearly organized structure*

At first glance, the structure on the left (Figure 2-1) might seem the best solution, since it is definitely faster (everything resides in a single directory), but in practice, it is surely the most discouraging and inconvenient setup for a person who has no previous experience with the project. Grouping all the palm tree files into one folder and listing them alphabetically doesn't impose any logical structure on the information. Because files used for different purposes are thrown together, the user must go through a process of trial and error, scanning every name and guessing at each file's purpose. Imagine doing that all day.

The structure on the right (Figure 2-2) makes it easy to understand where to find all the files needed for the project, and their purpose. Files are grouped together by how they will be used and arranged hierarchically. Just like your object-oriented code, this kind of file structure is logical and depends on the relationships among assets. It takes a bit of extra thought to set up your structure this way, but the investment is worth it in time saved and frustration avoided. Even someone unfamiliar with the project could pinpoint a specific file in no time with this kind of organization.

Figure 2-3 shows the basic structure I usually use. The elements are

- Root: The root of your project
- Map name: The name of the layer where you'll put the models and textures for your objects
- Obj name: The name of the 3D object
- FBX: 3D model export in FBX format
- MAX: Source file of the 3D model
- PSD: Source files used for this model
- TGA: Exported texture files in TGA format used for the 3D model

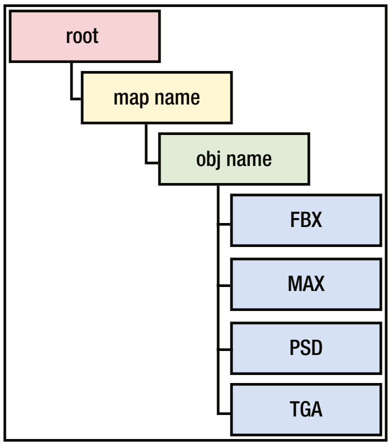


Figure 2-3. *A basic folder structure for a common 3D project*

You can expand on this scheme, of course. For example, if you have video files, Adobe After Effects projects, and sequences rendered from 3ds Max, you can add folders, as shown in Figure 2-4.

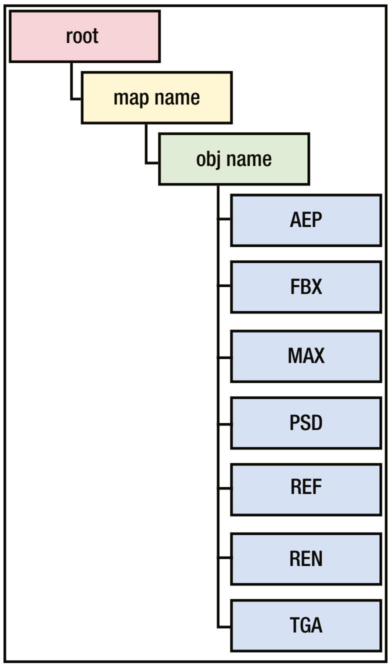


Figure 2-4. *The basic folder structure expanded to a more "render-oriented" setup*

Sometimes you will have items such as textures or render sequences of images that are shared by multiple projects. In this case, you should use a common folder (see Figure 2-5) to store your shared content. Or, opt for the most expensive choice (in terms of space) but more convenient because it creates fewer dependencies: duplicate your content in the folders of the projects where they are used. For example, if the object `STA01_palmTree03` shares one or more textures with the object `STA01_oakTree01`, the textures would be found in the folders of both objects.

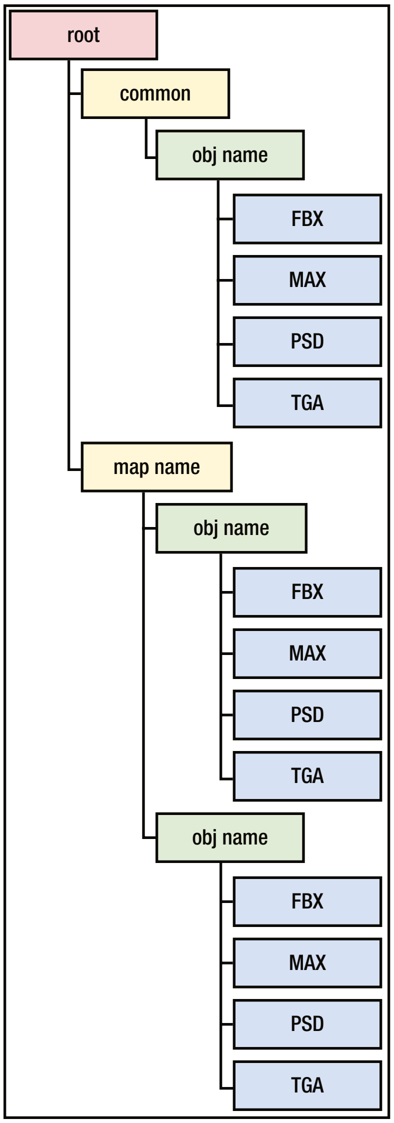


Figure 2-5. *The complete folder structure of the project, also with the common tree*

Avoid using linked references to other projects, as this practice usually creates confusion, even though it might save on disk space (which really isn't a big problem anymore).

Naming Conventions

When naming files, I usually use a system that includes the name of the asset and a prefix indicating the name of the map to which it belongs. You can choose the number of characters to use; my advice is do not overdo it and be as brief as possible without having too many restrictions. Since most projects contain a vast amount of assets, in order to avoid a "wall of text" effect, it's very important to maintain a very short, easy-to-read naming convention. A prefix with a length of 3-5 characters is ideal.

3D Models

When you are working with 3D models, always save both source files and exports; never keep only the export because it may contain only a reduced version of your model. For example, some scripts, rigs, or animation content might not be saved in the export file.

However, in either case, the naming conventions are the same. Suppose that you're working on a map that has a setting for a train station. The suffix might look something like this:

```
STA01_palmTree03
```

This sample suffix is organized as

- <map name>_<object name><incremental number>, where
  - <map name> is the name of the map that shows the location of the object (`STA01`).
  - <object name> is the name chosen for the object (`palmTree`).
  - <incremental number> is the number of the object version (for example day/night state or split into two chapters) (`03`).

You may also need to add a category for the object. My suggestion is to keep it short (three characters are enough in most cases), as in

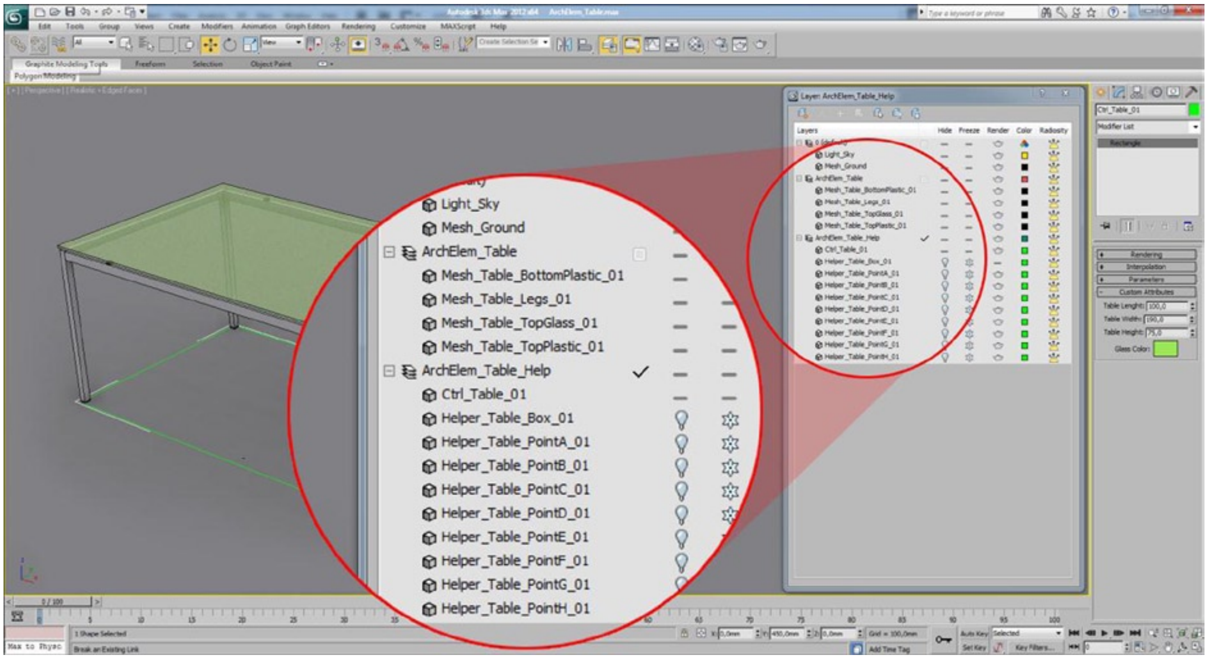
```
STA01_VEG_palmTree03
```

As you can see, most of the suffix is the same as before, but I have inserted the category after the map name. This suffix breaks down as follows:

- <map name>_<category>_<object name><incremental number>, where
  - <map name> is the name of the map where the object is (`STA01`).
  - <category> is the category of the object, in this case the abbreviation of vegetation (`VEG`).
  - <object name> is the name chosen for the object (`palmTree`).
  - <incremental number> is the number of the object variation (`03`).

In addition to constructing solid file naming conventions, you should create a clean and well-organized internal structure of your `.max` file.

In Figure 2-6, in addition to the 3D model, there is a mini-rig to procedurally control certain properties, like the three measurements of the table (height, width, and depth) and the color of the glass material. Layers are used to define which visual objects are part of the mesh, and which are control objects to help the mini-rig.



Light_Sky
Mesh_Ground
ArchElem_Table
Mesh_Table_BottomPlastic_01
Mesh_Table_Legs_01
Mesh_Table_TopGlass_01
Mesh_Table_TopPlastic_01
ArchElem_Table_Help
Ctrl_Table_01
Helper_Table_Box_01
Helper_Table_PointA_01
Helper_Table_PointB_01
Helper_Table_PointC_01
Helper_Table_PointD_01
Helper_Table_PointE_01
Helper_Table_PointF_01
Helper_Table_PointG_01
Helper_Table_PointH_01

Figure 2-6. *An example of a naming convention for a model with rig, inside 3ds Max*

The naming convention here is slightly different from what I've described above, but the goal is the same: to immediately understand the purpose of the objects in the scene.

Textures

For textures as with 3D models, you must always maintain the source files together with the exports. Image file names and folders need to be as clear, simple, and well organized as those for 3D models.

Texture naming conventions are based on the characteristics of the textures. Names include the kind of object, the type of material, and a suffix that indicates the channel type in which they will be applied.

I use the following abbreviations (see Figure 2-7):

- _D: diffuse map
- _S: specular map
- _NM: normal map
- _HM: height map or displacement map

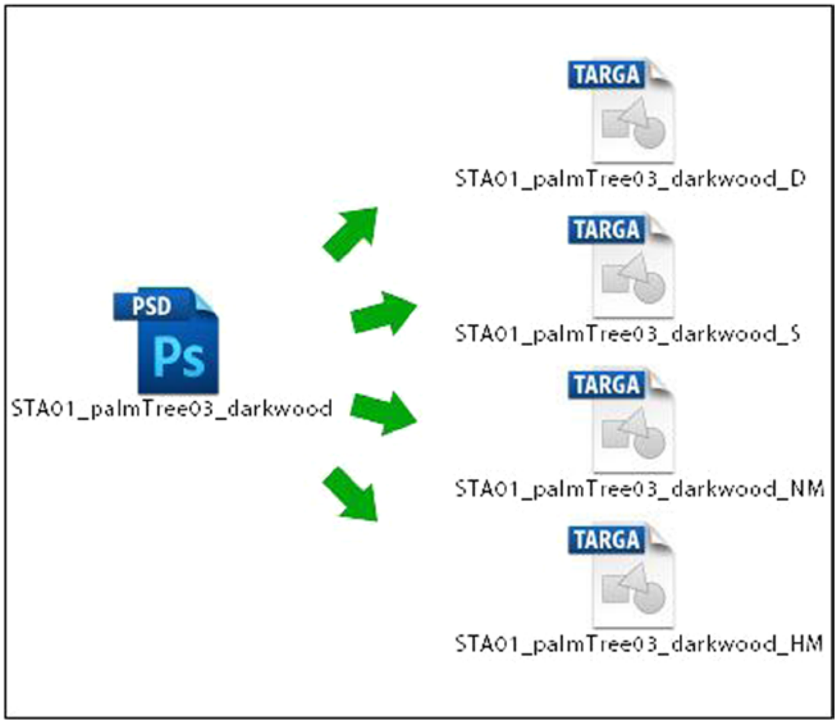

PSD
Ps
STA01_palmTree03_darkwood
TARGA
STA01_palmTree03_darkwood_D
STA01_palmTree03_darkwood_S
STA01_palmTree03_darkwood_NM
STA01_palmTree03_darkwood_HM

Figure 2-7. *From a single .psd file, you will create the textures for all the channels (diffuse, specular, normal map, and height map)*

In Figure 2-8, you can see that some folders are used to divide the layers according to their channel usage when exported, in order to group the parts that require a combination of multiple layers and layer colors in order to help the division within the folders.

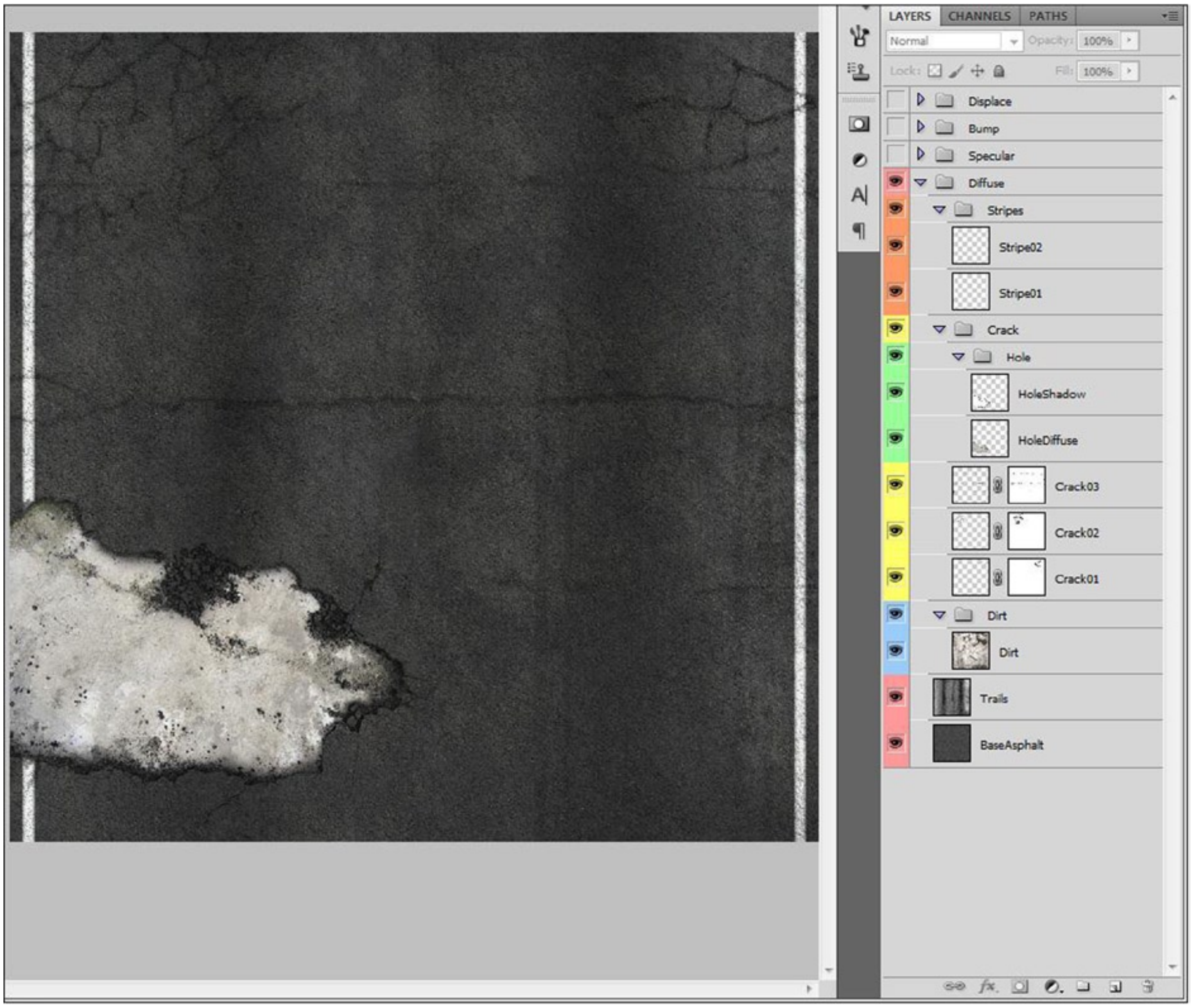


Figure 2-8. *An example of organization within a Photoshop file. The folder and the different colors help the reading of the file*

You might also want to categorize by Smart Object, which introduces the concept of instances inside Adobe Photoshop. Modifying one Smart Object updates all the linked instances, which means that you can use that feature to easily change all the buttons of your ingame menu, or apply nondestructive transformations to some layers without losing your original image data. For more information about the Smart Object feature, see `http://help.adobe.com/en_US/photoshop/cs/using/WSB3154840-1191-47b7-BA5B-2BD8371C31D8a.html#WSCCBCA4AB-7821-4986-BC03-4D1045EF2A57a`.

Trying to maintain multiple versions of the file you're working on, or incremental variations of the file name, is often a cause of chaos. At best you can lose time searching for a particular version; at worst you can make intricate changes to the wrong file, hold everyone up, and miss deadlines.

To solve this problem, you need a versioning system, such as Perforce, which will give you a revision history. With such a system in place, you can always roll back to a previous version, and you will solve three problems:

1. You always have the latest version of the file available.
2. You can return to any version you want at any time.
3. You will have a consistent backup of your data.

Of course, an infrastructure with Perforce is not for everyone. A good alternative I use every day is a cloud system like Dropbox or SugarSync, which provides enough functionality for most situations. GitHub is another popular version control system.

Conclusion

In this chapter, I wanted to show the organization system that I most frequently use for my projects. However, this is not the only way; it's just one possibility. The most important thing is clarity; without it, the quality of the work will undoubtedly suffer.

Some small rules that should be never forgotten:

- Always write in English, so anyone can understand.
- Spend the required amount of time to organize your work and your data.
- Making your work easy and clear will help not only you, but also anyone who takes over the project.

PART 2

Geometry and Models