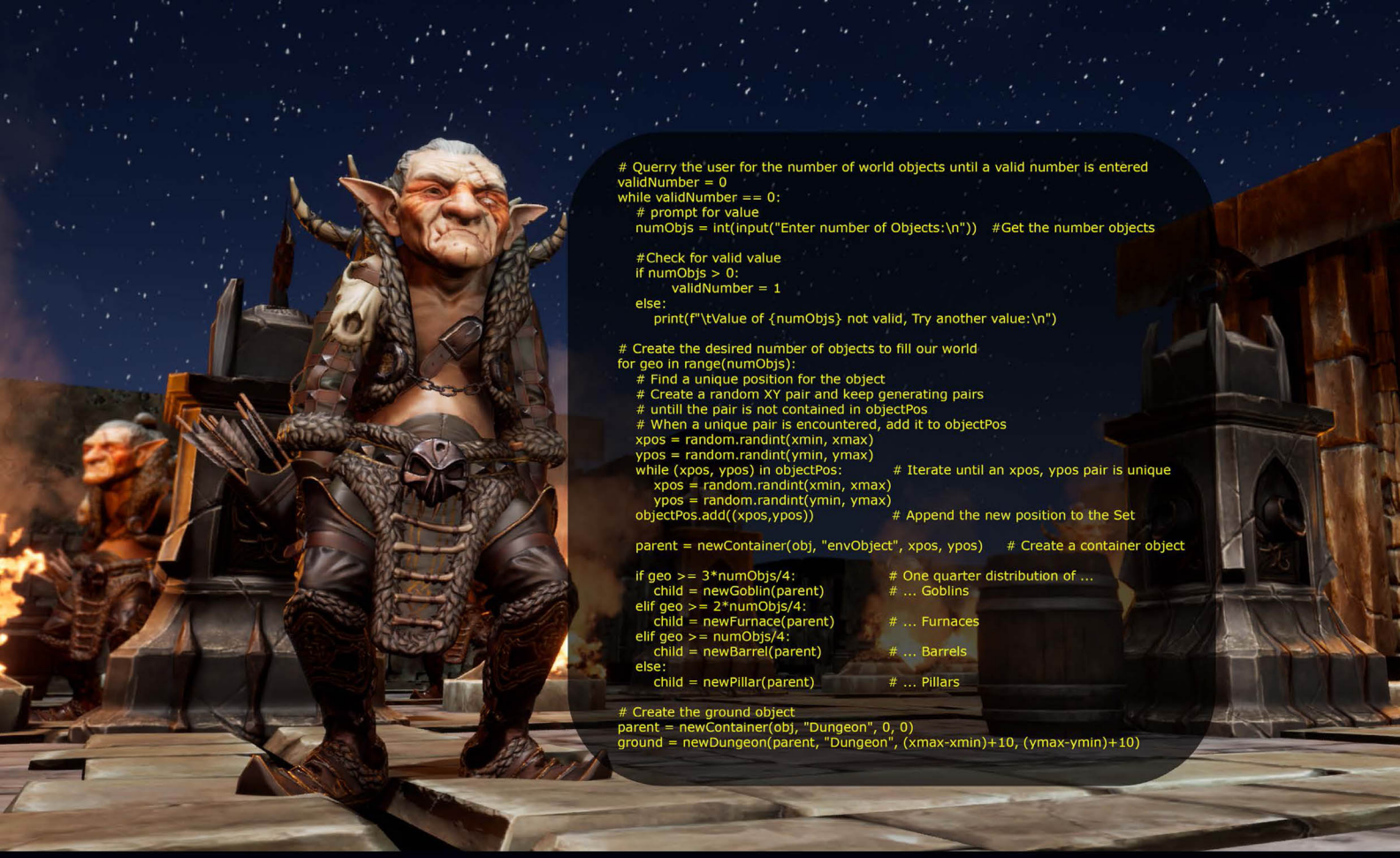

Essential Programming for the Technical Artist

Chris Roda

CRC Press
Taylor & Francis Group

ESSENTIAL PROGRAMMING FOR THE TECHNICAL ARTIST

This book is based on a successful curriculum designed to elevate technical artists, with no programming experience, up to essential programming competency as quickly as possible. Instead of abstract, theoretical problems, the curriculum employs familiar applications encountered in real production environments to demonstrate each lesson.

Written with artists in mind, this book introduces novice programmers to the advantageous world of Python programming with relevant and familiar examples. Any digital artists (not just technical artists), will find this book helpful in assisting with day-to-day production activities.

Concentrating upon subjects relevant to the creation of computer graphic assets, this book introduces Python basics, functions, data types, object-oriented programming, exception handling, file processing, graphical user interface creation, PEP 8 standards, and regular expressions. Programming within the SideFX Houdini 3D animation software provides a familiar environment for artists to create and experiment with the covered Python topics.

ESSENTIAL PROGRAMMING FOR THE TECHNICAL ARTIST

Chris Roda


CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Designed cover image: Chris Roda

First edition published 2024
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton, FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-0-367-86009-7 (hbk)
ISBN: 978-0-367-82040-4 (pbk)
ISBN: 978-1-003-01642-7 (ebk)

DOI: 10.1201/9781003016427

Typeset in Futura
by codeMantra

Dedication

To my best friend, Ha. From day one, you are the most important thing in my life.

Contents

Acknowledgments

This book would not have come into existence if it were not for two remarkable technical artists: Pat Corwin and Chris Lesage. These are two artists whom I have never met in person. They are the brave individuals who answered my reviewer requests posted on https://tech-artists.org. To say that their contributions to the project were significant would be an understatement.

Their patience and steadfast persistence when reviewing arduous chapters (especially dealing with object-oriented programming) were remarkable. Through the most challenging of criticisms, they remained professional, respectful, and encouraging. They are true technical art paragons: they do whatever it takes to deliver outstanding content through creative pipelines and make artists feel wonderful about the process.

Pat's technical understanding of the Python language knows no bounds. Yet he also understands how to use correct verbiage required to communicate core messages. His personal experiences and suggestions were invaluable when evaluating lesson effectiveness.

Chris' mastery of technical language is unparalleled. He identified language inconsistencies and provided valuable, alternative suggestions for incoherent content. He streamlined information flow by pruning irrelevant material and calling out missing details.

Thank you, Pat and Chris! Working with you gentlemen enlightened me to many new concepts and perspectives. I feel I am the student. I did my best to accommodate to all of your suggestions. Your contributions made this a far more effective text. I am lucky to have you on my team!

Finally, I would like to acknowledge SideFX, the creators of Houdini. I suppose it is obvious I am more than just a bit biased about this software. I have had the honor of collaborating with some of the brightest minds in 3D computer graphics. Houdini and SideFX were my most consistent teammates throughout my career of almost three decades. I could not have asked for more cooperative partners. Kim Davidson and SideFX team has always been there to assist in any way possible. Thank you for your consistent support.

About the Author

Chris Roda is a Technical Art instructor at the *Florida Interactive Entertainment Academy* (FIEA), a graduate degree program in interactive, real-time application development at the University of Central Florida. Early in his career, Chris was a visual effects artist in the film and television industries where he contributed visual effects for films such as *Spiderman*, *Titanic*, and *Fifth Element.* Before coming to FIEA, Chris was a CG Supervisor at Electronic Arts, where he worked on video game titles such as *NCAA Football* and *Madden NFL Football*. In addition to teaching, Chris works on generating tools and pipelines for the creation of *digital immersive experiences*, the amalgamation of the narrative of films, the interactivity of video games, and the immersion of theme parks.

INTRODUCTION

Programming is an essential component to becoming a world-class digital artist. This applies to technical artists and any artists with the ambition of leading practitioners of their fields. They all need to learn some kind of programming. Mastery is not essential but familiarity is. The world's greatest painters need to be familiar with some chemistry to master their pigments. Likewise, digital artists need to be familiar with programming: communicating with computers.

Computers are powerful tools. They can accomplish anything they are instructed to do. The challenge is instructing them in their proper language. Digital artists do not need to like talking with computers. They just need to be comfortable with them. This book does not substitute traditional, computer science texts which provide full topic coverage. The information covered in this book is adequate to introduce all technical and digital artists with essential programming skills necessary to tell computers what to do.

WHY THIS BOOK?

While working in the real-time game industry, I encountered a wide variety of technical artists with a wide variety of programming skills. Many were familiar with scripts which they had had copied from web-based tutorials. They were familiar with the desired end results and a little about input. However, they possessed only vague understandings of how the scripts started with input and produced output. These fragile script relationships were problematic when errors inevitably arose, or they required customization to satisfy specific situational needs.

As a computer graphics supervisor, I was aware that many technical artists could fly under the radar of programming accountability by scouring the web for relevant information. This strategy possessed small value when extrapolation or lateral thinking was required. To remedy this situation, I created a base curriculum providing technical artists with fundamental programming understanding which would aide them to understand any scripting situation. They would still be able to copy scripts from the web, but now they could understand how they were accomplishing their results and could easily modify the existing code to satisfy their needs.

This book is the result of that curriculum. Coming from a traditional programming background, I realize that this book's contents are a subset of what a full computer science course would cover. I also recognize core topics necessary for a technical artist programming toolkit. This text covers only those topics essential for production of computer graphics. It is folly to believe that one text covers all technical artist programming demands. However, I believe this book provides fundamentals required for entry-level technical artists to begin learning and understanding how to instruct computers to assist in creating digital art.

INTENDED AUDIENCE

As the name of this book implies, this text was originally created as a technical artist programming resource. The term "technical artist" is redefined by each development company. Technical artists tend to fall within four principle responsibility domains: pipelines, rigging and animation, lighting and rendering, and visual effects. There are a plethora of sub-domains and discipline fusions, enough to fill other books with their variations. The contents of this book should be adequate assisting with the primary four domains and their unlimited variations.

Pipeline artists utilize programming every day. Custom scripts are essential for the shepherding and conditioning of the billions of art assets required to fuel the real-time industry every day. Scripting provides oxygen for rigging

DOI: 10.1201/9781003016427-1

and animation technical artists. What requires weeks to accomplish by hand is now expected to be achieved in hours by competent riggers. While lighting and rendering appear to be pure artistic expression, they quickly become the most complicated and demanding coding environments. Familiarity with rendering, shading, and material principles is required for even the most artistic of art directors. (See *Lighting and Rendering for the Technical Artist*.) Many visual effects artists believe they can get by without programming. It does not take long for clients to demand visual output outside of what default interfaces provide and the need for custom code becomes essential.

All real-time artists and designers benefit with some programming foundation. Character, prop, and environmental 3D artists use scripting for creating, conditioning, and moving their assets to facilitate faster, in-engine verification. Animators are dependent on code to not only create their character rigs and controls but also to wrangle and manipulate billions of animated channels stemming from motion capture and key-frame animation. Designers make their living wielding procedurality: a discipline leveraging the power of programming to create infinitely more than achievable by hand. The days of designing scenes and levels are over. Modern real-time applications demand creation of entire worlds.

HOW THIS BOOK IS ORGANIZED

This book is broken into 20 chapters. Anxious readers desiring to get right into Python programming should jump directly to Chapter 6. Chapters 6–14 cover using Python within the Houdini context and focus on specific Python-related topics. Chapters 15–20 explain topics through the Python perspective but are applicable to any programming language. The first five chapters address the philosophical aspects of this book: relationships of technical artists with programming, why Python is the prime focus of this book, and a history of how this book came into being.

- **Chapter 1: Introduction** provides a quick explanation of how and why this book came into being, and how it is organized.
- **Chapter 2: The Role of Programming in Technical Artist Life** discusses technical artist roles and their relationships with programming.
- **Chapter 3: Programming Strategies** offers fundamental technical artist programming philosophies.
- **Chapter 4: Computer Languages** surveys modern programming languages and explains why technical artists should learn Python.
- **Chapter 5: Programming Inspiration** describes the six-week curriculum this book is based upon. Justification for using Houdini as the Python programming environment is provided.
- **Chapter 6: Python Setup and Orientation** provides instructions for Houdini installation, accessing Python, and orienting readers with Python functionality.
- **Chapter 7: Python Basics** explains how Python handles string manipulation, commenting, and expressions. Example scripts get programmers creating technical artist-like tasks.
- **Chapter 8: Python Logic** covers Python script data flow. Explanation of code blocks, conditional statements, and looping is provided.
- **Chapter 9: Python Functions** explains Python function integration. Instructions for handling external functions and defining custom functions are provided.
- **Chapter 10: Python Data Types** covers the different types of data intrinsically included in all Python installations.
- **Chapter 11: Python Object-Oriented Programming: Foundation** introduces working with and authoring Python object-oriented programming classes.

- **Chapter 12: Python Object-Oriented Programming: Inheritance** describes how Python classes can take advantage of class derivation and composition.
- **Chapter 13: Python Exceptions** explains how Python deals with special situations, including errors and warnings. The Python *try-except* protocol prevents special situations from interrupting program flow.
- **Chapter 14: Python File Processing** demonstrates how Python reads and writes to external files. Instructions and examples of *JSON* and *XML*, and pickling file formats are provided.
- **Chapter 15: Command Line** introduces readers to the concept of interacting with Python outside of graphical user interfaces. This style of interfacing is useful when managing day-to-day pipeline operations.
- **Chapter 16: Python Graphic User Interface** describes graphical interface tools available to Python. In-depth instructions are provided for getting users familiar with the *Pyside 2* interface toolkit.
- **Chapter 17: QtDesigner** provides instructions how to use *QtDesigner* to design PySide and PyQt interfaces.
- **Chapter 18: Python PEP8 Standards** describes the PEP8 standards; conventions encouraging efficient, effective, and readable Python code.
- **Chapter 19: Regular Expressions** introduces readers to *regular expressions*; formalized, textual pattern recognition strings used for identifying patterns within input strings.
- **Chapter 20: Conclusion** concludes this book and provides a few words of encouragement.

CONVENTIONS

This text presumes that the majority of readers do not come from technical backgrounds nor necessarily have affinity for the technical field. Regardless of background, this book treats all readers as technical artists, folks who create art using technology. Hopefully, this definition encourages the greatest number of people.

There may exist some confusion between *technical artist* and *technical director* roles. In the broadest perspective, *technical artists* and *technical directors* share almost identical responsibilities. Their skillsets are similar. The big difference between the two is that technical artists deal with real-time art pipelines and technical directors deal with off-line renderers. Real-time applications include video games, simulations, and immersive experiences. In general, technical directors who operate off-line renderers are more concerned with quality and less concerned about performance. This differentiation becomes fuzzier every day as teams use real-time engines for creating off-line, film-quality cinematics, and traditional off-line companies use the same real-time engines for virtual productions.

Confusion may occur when labeling *real-time technical directors*. *Technical directors* within real-time environments are experienced programmers responsible for implementing all technologies required for a successful project. The role should not be confused as an artist appointment.

SOFTWARE DISCLAIMER

Houdini, from SideFX, is used as the demonstrational Python environment. There are two motivations behind this decision. This first is simply the author is most familiar and created the curriculum within the Houdini environment. The second reason is that Houdini Python is very object-oriented and there is clear separation between the Python code and the Houdini graphics *application programming environment* (API). Motivated readers are highly encouraged to replace Houdini portions with APIs from their favorite software. This exercise reinforces the Python lessons and bolsters understanding of the Houdini and other software graphics APIs. The author is happy to assist with any questions dealing with future conversions.

PREREQUISITES

The only real prerequisite for this text is the open mindedness to learn. While not apparent, any attempts to absorb this material will have exponential results. The examples in this text were taken directly from a curriculum dedicated to teaching basic programming to novice technical artists. Even simply transcribing the examples establishes muscle memory that cannot be unlearned. Playing with and ultimately breaking the examples accelerate the learning process. Continuous diligence to this discipline provides consistent returns. Like going to the gym, results are gained while there is work. All students who follow this curriculum achieve some sort of programming comfort as long as they are willing to try. While some take faster to the material than others, all are able to apply this book's contents to their daily activities. The author is confident that you will too. Enjoy the process.

INTRODUCTION

The foremost responsibilities of technical artists are to do *whatever* is required to assist artists' delivery to real-time experiences such that the art's integrity is not challenged or the stability of the experience is not jeopardized. *Whatever*, often translates to performing tasks that are unpleasant and potentially confusing. Examples include updating file paths from one version of an iterative project to the next, retargeting all animation clips from one character to another, or building tools for generating inverse and forward kinematic rigging for arbitrary characters of various sizes and proportions. Sometimes these responsibilities include achieving rapid results devoid of creativity such as copying and renaming thousands of files to satisfy spontaneous marketing requests. Often technical artists get by with brute force or performing each task step by hand. However, the time demands of most situations typically dictate the need for programming or scripting solutions.

The context of the situation has significant impact on the technical artist's decision to program or not. Once the programming decision has been made, the technical artist must decide what tasks the tool must achieve, its stability and how interactive it is. Figuring out the tool's root demands are essential for understanding clear tool solutions and requirements. Many of these decisions are dependent on the artist's role within her organization. Other times, these decisions are made by her disciplinary context. For example, a quick script traversing a character rig to search for misspelled joints is dramatically different from a tool for reliably relocating project files within version control. A technical artist's programming life is made significantly easier when the programming expectations are understood in advance.

TECHNICAL ARTIST PROGRAMMING RULE #1

The first rule of technical artist programming is, "When a task is to be repeated more than twice, a script should be written". Of course, there are exceptions to this rule and they are dependent on "return of invested time" required to create the tool. For example, if five animation clips require keyframes at frame -10, then a script probably would not be worth the invested time. When the investment of time required to create the script is significantly greater than the time the tool will save, the script is not worth creating. However, as in the previous example, the same keyframe needs to be created for hundreds of characters and thousands of animation clips; a script should probably be created. Tools that satisfy a need only once are called *one-offs*. One-offs should generally be avoided unless they perform prohibitive tasks such as populating a unique stadium with thousands of fans.

EXPECTATIONS OF A TECHNICAL ARTIST PROGRAMMER

When working within the context of a professional development environment, technical artist programmers will take on one of two roles, a guerilla problem solver or tools programmer. Their programming expectations are dependent on their role.

Guerilla Problem Solver

When functioning as a guerilla problem solver, technical artists must first consider practicality and then make sure that the code is clean, safe, and functional. The code does not necessarily need to be efficient or robust. The code only needs to achieve the requirements of the task and be written and employed within the shortest time period. Standard questions of interface design should be considered: *what*, *who*, *where*, and *when*.

DOI: 10.1201/9781003016427-2

"What does the tool do?" is an important consideration for keeping the code in scope. Within the context of this question, the code should perform only one task which its name implies. When the tool does more than the one task, the opportunity for miss-use and redundancy escalate.

"Who is using the code?" is a question which has a strong impact on the expectations of the code. When the code is written only for the coder to use, the code may be a bit rougher than when written for other users. The coder has an intimate relationship with the tool and understands its eccentricities. When written for other users, the code needs to be logical, the names need to make sense, and there should be robust documentation. When written in this fashion, others examining the code should be able to quickly and easily understand what the code is doing and how it is doing it. Even when the code is written by technical artists for personal use, they do forget things and it is good practice to write code to enhance future understanding and prevent confusion.

"Where the code is going to be used?" is also an important question. Does the code function as a stand-alone tool? Does it work within the context of a *Digital Content Creation Tool* (DCC)? Does it complement the asset conditioning pipeline? Stand-alone tools tend to be less demanding while DCC and pipeline scripts need to be written with understandability, re-use, and maintainability in mind.

Lastly, "When does the tool need to be done?" has a direct impact on code quality. Code created to put out today's raging fire is going to be rougher than code scheduled with ample planning time.

The *Minimum Viable Product*, *MVP*, of a script is the minimal number of tasks the code is expected to accomplish. Guerilla programmers must only provide the minimal amount work required to achieve the MVP. Focusing on the MVP maximizes technical artists' time and minimizes the time to get into the hands of the end-user. Focusing on code to contribute to the MVP reduces the amount of superfluous work and minimizes time spent on duplicate or redundant code. The need for the code to achieve its MVP is higher than the need for the code to be efficient and clean.

Suppose a game team suddenly pivots to create a crowd scene environment in order to take advantage of recently acquired *Motion Capture*,(MOCAP), data. A cache of 50, pre-modeled characters is available but none have skeletons compatible with the data. The marketing team needs footage of the scene to meet an important deadline. The team rigger decides to create a tool to semi-automate the skeleton creation process for the characters which reduces the rigging time from weeks to a day. Since the characters are to be animated using specific MOCAP data, there is no need to include other typical rigging functions such as IK/FK switching, spline spines, or reverse foot IK. Since this tool is a *one-off* and will only be used to rig these specific characters for the MOCAP data, the script writer focuses less on efficiency, readability, and re-use and more on rapidly completing the task to meet the marketing deadline.

In the prior example, the lack of a robust, efficient, and re-usable functionality is no permission for the code to be intentionally unpolished. However, technical artists should devote no more time than is essential for accomplishing the MVP and then move on.

Tools Team Member

When technical artists shift from being guerilla problem solvers to being a members of tools teams, their programming expectations make a radical shift. As guerilla problem solvers, technical artists create minimal amounts of code accomplishing only immediately needed tasks. As members of a tools team, technical artists produce tool experiences which not only satisfy necessary needs and requirements but also generate positive feelings of accomplishment. Creative focused tools must generate "the warm fuzzies" when used. Most often, creative artists are the users of technical artist tools. Artists respond more favorably to tools which empower them to feel creative over tools which behave logically. As artists, technical artists are equipped with the creative ability to code posi-

tive emotional experiences. Team member coding expectations must not only satisfy the best software engineering traditions but also appeal to the artist experience.

One of the most fascinating models of technical artists creating tools for artists is Deep Silver Volition studios. At Volition, the programming team develops and maintains the core real-time engine and exposes functionality in a python API. Technical artists take the API and develop tools for the artists and designers. With this model, technical artists focus on creating tools and interfaces dedicated to the artists, keeping them happy while freeing programmers to generate effective and efficient engine code.

When working for a tools team, technical artists must create tools which are "artist-proof". In other words, interfaces must persist so that no artist input, intentional, or unintentional, can prevent the tools from functioning as expected. Tool makers should think and behave like artists to create interfaces generating warm fuzzies and anticipating workflow barriers which hinder artistic experience. As a general rule of thumb, any time a tool crashes or interrupts creative workflow, regardless of programming error or un-anticipated artist input, it is the tool maker's responsibility to make sure the situation cannot repeat.

A tool must be stable while satisfying its MVP while being robust enough to provide artists with enough freedom to easily customize output and satisfy creative intention. Because a tool is created within the context of the programming team, the tool must solve a unique problem which has not already been solved by another tool. Its code must be well documented for explaining algorithmic intention, future maintenance, and re-use. The code itself must be efficient and stable while structured for future expansion and integration. Unpolished and unprofessional code is unacceptable.

DISCIPLINE-BASED EXPECTATIONS

The programming expectations imposed on guerilla problem solvers and tools team members are different and so too are the needs of the various technical art disciplines. The following sub-section describes of the fundamental responsibilities for each of the technical art types: pipeline, procedural world building, rigging, animation, technical animation, rendering and lighting, and visual effects.

Pipeline

Programming skills are essential for pipeline technical artists. Almost every activity involves digital asset processing: conditioning, transportation, version control, and troubleshooting.

Conditioning

Translating digital art assets into formats which real-time engines understand is a programming art form. Commercial real-time engines such as Unreal, Unity, and Lumberyard are amazing. Remarkable attention has been devoted to making the experience of translating digital art assets into engine compatible formats as easy and painless. No two real-time engines are the same. Larger production companies mostly employ in-house engines or customized commercial engines. Unique engines impose conditions upon art assets so they may be imported successfully. The requirements increase in proportion to the uniqueness of the real-time engine. Many engines venture outside the familiar *FBX digital intermediate* format when importing content. (Thorough details on this topic are covered in *Technical Artist Pipelines and Workflows*.)

When digital assets fail to meet these requirements, the import process fails and often causes engine failure. Sophisticated tools condition art assets for successful engine import. It is the pipeline technical artist's responsibility to build, maintain, and expand these conditioners to meet each team's production needs. This is a full-time production job and technical artists devote significant portions of their day simply addressing production emergencies. When asset conditioning pipelines work efficiently, they are transparent. However, when broken they represent huge losses of expensive time.

Transportation

Data wrangling is a crucial component for maintaining stable asset pipelines. Clean and orderly data flow contribute to a project's profitability. Technical artists must collaborate with the programming and production staff early in project pre-production to establish standards which the entire team needs to follow. It is often the role of the technical artist to set up these standards as well as enforce them.

Data file organizational structure is essential for maintaining team communication. Lost assets are a common occurrence. Valuable production time is lost in the search for these assets. Establishing and enforcing directory structure dramatically reduce the number of misplaced assets. Successful structure also ensures future file communication. Assets fall into logical slots without the need for explanation or redirection. When assets reside where they are expected and are found without additional clarification, digital asset pipelines run smoothly. Listing 2.1 is a hypothetical example of a typical art asset file structure.

```
Art Assets
  Characters
    Character A
      Mesh
      Skeleton
      Animation Clips
      Materials
        Textures
    Character B
      ⋮
  Environments
    Environment A
      Mesh
      Materials
        Textures
    Environment B
      ⋮
  VFX
    Mesh
    Skeleton
    Sim Data
    Materials
      Textures
  Materials
    Textures
```

Listing 2.1 Typical Art Asset Structure

Every production company has its own custom structure designed to satisfy its unique needs. One structure is not necessarily better than another as long as consistency is maintained. When the structure is maintained and defended, it behaves as intended.

Digital asset naming conventions are similar to file structures. Naming conventions provide the essential information required to organize, classify, and identify any assets exclusively through the structure of their name. Naming conventions identify individual file categories, types and unique descriptions. Adherence to naming conventions is crucial for smooth production workflow. At any time, teams may wish to alter their convention to better suit their needs. However, changes made later into the project are more challenging to implement. Initial time invested at the beginning of a project from the lead technical artist, designer, and programmer will pay off in the long run.

The following example is a naming convention template for generic digital assets:

$$\langle Category \rangle\_\langle Asset \rangle\_\langle Item \rangle\_\langle Detail \rangle\_\langle Type \rangle$$

Listing 2.2 contains example names conforming to this convention.

```
Mesh_Bink_Torso_Hi
Texture_Painesville_Firehydrant_1K_Diffuse
Simulation_BridgeCollapse_VersionA_Long
Material_Firehydrant_Worn
```

Listing 2.2 Typical Asset Names

All abbreviations, nicknames, and codes should be established in the naming conventions. Their effectiveness is dependent directly to their adherence.

Version Control

Most organizations and production teams employ some sort of *version control software* (VCS). VCS systems manage all changes to computer programs, data sets, digital assets, and any documents which may be needed for future reference. The software systems make historical digital backups of its documents and allow retrieval at any point along the document's history. Digital production teams rely heavily on RCS systems to prevent catastrophic information loss and minimize the amount of overlapping work.

While VCS systems are essential production tools, their cumbersome interfaces make them unpopular with artists. It is the technical artist's responsibility to guarantee artist participation within the system. This task often includes creating easy-to-use tools to register assets with the system, modify the assets, rename, move, and delete the assets from the system.

Registering

Registering a digital asset with a VCS system makes the system aware of the asset's location and assigns system control over the asset. This is the first and most important operation for the artist to comply with the system. It is the technical artist's responsibility to implement workflows and tools for easy and painless asset registration. Assets which do not get registered with VCS tend to be lost or forgotten.

Updating

When digital assets need to be edited, modified, or otherwise updated they must first request control from the VCS system. The system will not allow modification to an asset until control has been formally granted. The artist is free to edit the asset once control of it has been granted. The artist also exercises exclusive control over the asset at this time. After modification, the artist pushes the changes to the server and relinquishes control back to

the VCS system. Initially, this process appears cumbersome. Technical artists provide tools to facilitate this process and minimize the opportunity for artist error. Technical artists must collaborate with the artists when creating tools which are not only easy and inviting but also empower the artists to register frequent changes with the VCS system.

Renaming, Moving and Deleting

During typical production, assets need to be renamed, moved, and deleted. These are precarious updates to the VCS system that must be handled sensitively as not all systems handle such manipulations uniformly and may require necessary operations to be executed predictably and safely. Most VCS tangled messes occur from improper handling of these operations. Technical artists significantly improve production workflow by providing interfaces and workflows which guarantee successful file renaming, moving, and deleting, and minimize operational mistakes.

Troubleshooting

Pipeline technical artists devote most of their time troubleshooting. Problems have the tendency of appearing in the most unusual places and circumstances. While these situations slow production, they do not clearly indicate where future tools and scripts are needed. Quite often, tools need to be created to help identify troublesome situations. For example, consider a situation of thousands of animation clips registered with VCS with incorrect naming which blocks all character creation and animation. Days of artist time are necessary to correct this situation by hand. However, a technical artist could create a script and resolve the situation within an hour. The technical artist is praised as a hero for the next few minutes or until the next troubling issue rears its ugly head.

Procedural World Building

Production teams are often presented with tasks of creating digital assets which are impossible or cost prohibitive to create by hand. Technical artists provide procedural world building tools to assist artists with the creation of these assets. Examples are cities, forests, world terrain, crowds, swarms and herds, very large objects such as galaxies and the very small such as sub-cellular viral infestations.

Technical artists employ proceduralism in their world building tools. Proceduralism is the creation of computer tools and systems which leverage the talents of artists to assist with creating digital assets which could not be achieved by talent alone. The concept of proceduralism is often misunderstood with automation. While automation requires no artist input, proceduralism is dependent on artist input which provides soul and life to the resulting assets. Outside of visual effects, procedural world building is the most creative endeavor for technical artists. The pursuit of proceduralism is often the gate through which artists enter into the realm of programming. More than randomness, proceduralism applies artist inspired filters to create intentional assets instead of hoping to find a few valuable pieces found in a sea of chaos.

While procedural world building tools are essential for creating assets too complex to be generated by hand, they are limited. Proceduralism is excellent for creating assets which provide volume to theme worlds yet are not the targets of participant attention. These are the assets which make the world feel genuine and complete. Hero, or attention drawing assets, should be generated by hand whenever possible. The intention of proceduralism is to reduce the costs associated with the creation of very large and precise data sets. When the costs of creating procedural systems to achieve specialized results exceed the costs of creating the assets manually, the later should be chosen. Artists are always capable of adding minute attention and detail where the cost of creating procedural systems cannot be justified. For example, consider a professional sports stadium. Procedural tools are outstanding for generating the facility structure. However, proceduralism cannot duplicate the unique statues decorating the environment. Proceduralism generates assets to fill the world and frees time for artists to create attention driving assets.

Rigging

Rigging is an essential component of bringing animated digital characters to life. Once a character model has been created and before an animator can start animating it, a technical artist, (called a rigger), must convert the model into a digital puppet. The core principles of rigging include creation of a skeleton, skinning the model to the skeleton, and creation of a control structure to manipulate the skeleton. Skeletons are the bio-mechanical structures, composed of bones or a sequence of joints, used for moving the character. Skinning is the process of mapping the character model to its relative bones and joints. For example, the character's left arm must be mapped to the corresponding bones comprising the skeleton's left arm. The control structure provides easy and accessible handles for the animator to quickly animate the character's bones. Technically, the control structure is not essential. However, most animators find animation of the character's skeleton difficult and challenging without the assistance of the controls.

The rigging process is an art unto itself and numerous books are devoted to the skills of the craft (see *Rigging for the Technical Artist*). While the fundamental primitive strategies are easily explained, the puzzles and challenges surrounding the implementation of unique character rigs are endless. An artist could devote an entire career focused to the task. The process at times can be tedious, frustrating, and prone to human error. When problems arrive, the debugging process may be more resource expensive than recreating the rig. This cumbersome task is remedied through the process of procedural rigging; the creation of tools and scripts which leverage the artistic skills of the rigger. This process requires creative input from the rigger to generate bio-mechanically correct movement. The skinning, joints and control structure for the character in Figure 2.1 were procedurally generated.

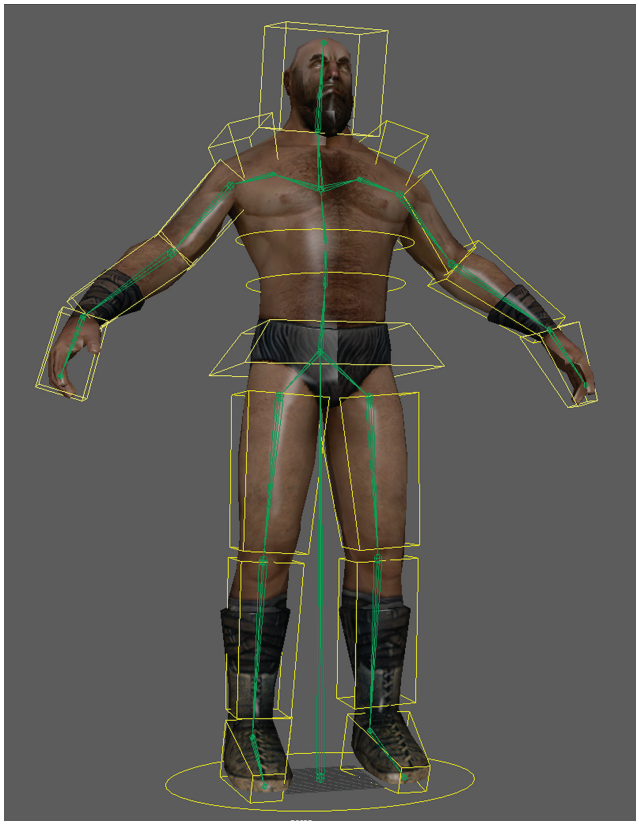

FIGURE 2.1 Character skin, joints, and control structure.

There is no standard set of tools used for rigging. Each technical artist must create her own set of tools and scripts which complement her individual style, wants, and needs. The process of creating procedural rigging tools feels intimidating at first. However, once completed, the tools make the fundamental process of rigging fast, reliable, and most importantly, predictable. Analysis of rigging code makes the process of debugging problems easier and faster than when done by hand. Rigging modifications are implemented by altering rigging code and regenerating rigs. Without the assistance of scripts, regenerating rigging can be an exhausting process. Over the course of production development, a single character rig may need to be regenerated hundreds of times. When the bulk of the rigging process is handled with tools and scripts, technical artists are allotted time for solving more complex problems which define the art form.

Lighting, Rendering, and Look Development

Most modern, three-dimensional, real-time rendering engines are equipped with node-based material editors. Material editors are front-end interfaces for sophisticated shaders which are often thousands of lines long. Written in dedicated shading languages such Open GL (GLSL), DirectX High-Level (HLSL), or Metal, shaders program the GPU's rendering pipeline. The rendering pipeline is the engine infrastructure that executes the shader's commands and produces visual output. Material editors are intended to be artist friendly and expose as many shader input controls as possible. However, when the desired visual output exceeds the material editor's exposed controls, programming solutions must be employed. There are two methods for interfacing with the rendering pipeline: manipulation of the pipeline code and modification of the shaders. Technical artists collaborate with the rendering pipeline through one or both of these techniques, depending on the artists' programming ability.

Pipeline Manipulation

Any modifications to the rendering pipeline's structure, algorithms, or the introduction of any new features require fabrications to be made in the pipeline's native language. In most situations, the language is C++. When programming within the rendering pipeline, a technical artist must take on the persona of a tools team member and be sensitive to the coding standards established by the pipeline rendering team. Effective communication with the team's Technical Director must be established to explain the technical artist's intentions. The technical artist must conform to the reviewing, testing, vetting, and documenting standards established by the team. Even when debugging engine rendering problems, special attention to communication with the programming team must be maintained.

Shader Programming

Instructions to the GPU's pipeline are implemented within the pipeline's programmable shaders. Modifications to the shader beyond exposed material editor input require programming in dedicated shading languages such Open GL (GLSL), DirectX High-Level (HLSL), or Metal. Real-time rendering pipeline shaders have seven stages: Figure 2.2 show the relationship between pipelines, shaders and materials. the input assembler, the vertex shader,

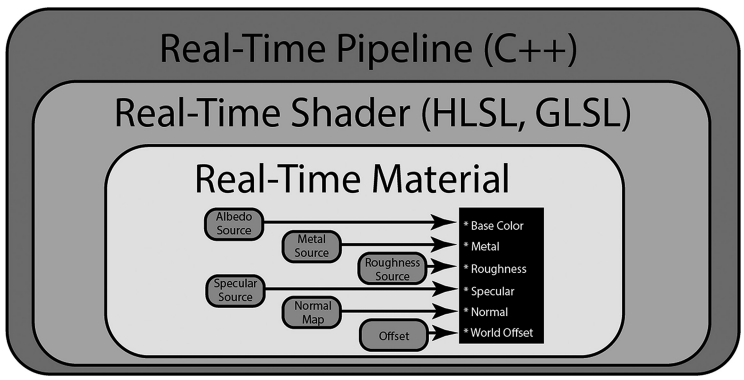

FIGURE 2.2 Real-time rendering pipeline hierarchy.

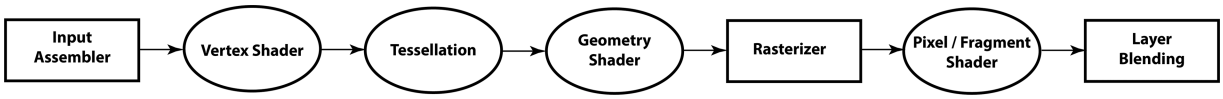

FIGURE 2.3 Real-time rendering pipeline stages.

the tessellation stage, the geometry shader, rasterization, the pixel shader, and the layer blending stage. Figure 2.3 shows the sequential order of the real-time rendering pipeline.

The input assembler collects the scene data from the CPU and organizes the data to be inputted to the pipeline. While not programable, this stage is configurable within the shader code. The vertex shader is programmable and configures all vertex attributes. Aside from manipulating all vertex data, the vertex shader's primary role is to transform vertices from object space to the appropriate camera projection space. The optional tessellation stage is programmable and instructs the pipeline how to subdivide the input geometry. The geometry shader, also optional, programs the pipeline to add or remove geometry. The configurable rasterization stage converts the three-dimensional geometry into a two-dimensional pixel bitstream. The programmable pixel shader provides instructions required to calculate color information for every pixel. A significant proportion of programming time is devoted to pixel shader programming. The configurable layer blending stage blends the recently rendered bitstream with prior processed layers. A thorough introduction to programming real-time shaders is given in the book *Lighting and Rendering for the Technical Artist*.

Visual Effects

Visual effects are specialized manipulations to the real-time engine and pipeline which manifest narrative enhancing phenomena to the real-time experience. Infinite in variety and approaches, visual effects are broken into four primary categories: in-camera effects, in-material effects, simulations, and particle effects. In-camera effects, such as camera flares, are performed exclusively within the last two stages of the rendering pipeline. While simple manipulations can be implemented within the material editor, more sophisticated strategies need to be programmed directly into the pipeline shader. In-material effects, as the name implies, manipulate all stages of the rendering pipeline and can be implemented within the material editor. However, extreme visual effect complexity must be handled in the pipeline shader. Elaborate editor materials maybe more computationally expensive compared to those implemented in the shader. Simulations attempt to duplicate natural physical phenomena. Any alterations to the physical simulation algorithms need to be programmed in the engine code. Technical artists programming in this context need to be aware of the conventions established by the engine programming team. Sophisticated modifications to the simulations' materials need to be programmed into the shader while simpler adjustments can be handled by the material editor. Particle effects are the combination of the three prior effect types: in-camera, in-material and simulations. Depending on the context, modifications to particle effects are made in the particle physical editor or in the material editor. More complex adjustments to particle simulation behavior need to be made in engine. Sophisticated adjustments to the particle's visual display may be programmed into the engine or into the pipeline shader. In-depth coverage of these visual effects categories is covered in the book *Real-Time Visual Effects for the Technical Artist*.

Technical Animation

Depending on their structure, certain production companies employ the talents of technical animators. Technical animators support keyframe animators by providing technical assistance beyond the capabilities of the DCCs at hand. Technical animators provide tools and scripts for assisting with the keyframe animation processes such as channel and keyframe editing as well as time manipulation. Tools may be required to assist with the MOCAP pipeline where the data require massaging, filtering, and cleaning. Custom tools assisting animation retargeting from one character to another are common. Most animation DCCs provide fundamental tools for supporting the animation and MOCAP processes. However, many tasks prove to be too laborious and time consuming to be performed without the assistance of additional technical animator tools.

Technical animators often aid with non-keyframe or procedural animation. They assist with real-time engine animation state machines and other procedural constructs which assist with the organizational blending of animation clips. Instead of being covered by a visual effects team, technical animators provide character related simulations. Secondary animations such hair and clothing animation, fat and muscle jiggle, and dangling accessory motion are generated with the aid of technical animator tools. As with other visual effects, DCCs are effective for generating basic behavior but anything beyond must be assisted with technical animator programming.

CONCLUSION

Too numerous to list, the types of tasks requiring technical artist programming skills are limited only by artists' imaginations. As a good rule of thumb, stick with the first rule of technical artist programming, "Any task that is performed more than twice should be replaced by a script or tool". The robustness of the tool should be determined by the context at which the tool is to be used. When the tool solves a quick and temporary problem, it can be rough and unrefined. When the tool is to be used by multiple individuals in a production setting, extra consideration must be made to make the tool user friendly, easy to understand, and easy to maintain.

There are six discipline-based topic areas where most technical artists need to program scripts or create tools for: pipeline, procedural world building, rigging, lighting, visual effects, and technical animation. Pipeline is the discipline where the largest proportion of scripts are created. The process of conditioning, shepherding, and troubleshooting game assets requires endless programming and tool creation. Procedural world creation relies heavily on dedicated DCCs but is enhanced significantly with additional programming. Character rigging is a demanding discipline requiring complex problem solving and is prone to human error. Procedural rig creation, driven by scripts, maintains repeatable techniques and reduces human error. The DCC user interface unintentionally limits user opportunity. Lighting, rendering, and look development require additional programming to reach beyond these limits. Many visual effects may be achieved through tools exposed through interface alone. However, as most clients desire original visual effects, programming and scripting are required to achieve unique results. Blending pipeline, animation, and visual effects, technical animators use programming to wrangle animation and MOCAP data and procedurally generate animated motion.

INTRODUCTION

Many technical artists do not have formal programming education. Many come from traditional fields like computer graphics, robotics, or artificial intelligence. Many are self-taught. They don't necessarily have knowledge about traditional software engineering. In the heat of production, these principles, while valuable, can feel distracting and time-consuming. This chapter covers strategies technical artists can employ immediately. Task identification effectively defines the scope and expectation of scripts and tools. Testing code as it is written is most often faster than debugging gobs of untested code. The dangers of blindly relying on resources and scripts found on the internet are also presented.

TASK IDENTIFICATION

Understanding exactly what a script or tool needs to do is the first step for creating effective code. When requested, technical artists must be careful to listen for what clients or teammates need and not necessarily what they are asking for. Often, requests call for specific techniques or strategies which sound essential but originate from insufficient technological understanding or lack of appreciation of applicable context. Effective communication is necessary for understanding vital tasks and strategies. Technical artist programmers must provide only what is needed: not too much and certainly not too little.

When at all possible, technical artists should initially perform requested tasks by-hand before implementing code. Working in this order has multiple benefits. Performing operations by-hand identifies repetitious patterns and sequences. Delegation of these patterns with code allocates more time for artists to focus on creative expression instead of repetitious procedures. This strategy educates technical artists how to successfully apply techniques. Often, they do not understand how necessary operations are performed, let alone implement relevant scripts. Executing by-hand provides ground truth evidence that solutions are achievable and endows them with knowledge of having successfully accomplished tasks at least once. Once there is understanding of how tasks are accomplished with respect to repeatable patterns, technical artists can start writing code.

ONE LINE AT A TIME

When writing code, an important strategy to adopt, especially by novice coders, is to write only one line at a time, followed by testing. This approach may seem counter-intuitive and slow. In the long run, it proves to be a faster and more stable strategy. Testing code after every written line has multiple benefits. The first is the code's correctness is verified. This functions as a granular type of unit testing. Establishing code effectiveness provides technical artist feedback and removes doubt of uncertain behavior. Identifying correctness also exposes bugs. Observing functioning code informs programmers where and when events happen so that they may be referenced in the future with minimal amounts of searching. The last benefit of this strategy establishes programming momentum. Often, coding feels alien and uncomfortable. Observing small amounts of success builds enthusiasm and whittles away doubt that hinders novice programmers.

Contrary to popular belief, programming is not magic. It is not something that simply flows through fingertips. Of course, there are a rare few for whom this is true. However, if you were one of them, you would not be reading this book. Novice programmers need to resist the temptation to write all of their code in one attempt and expect it to be functional upon first execution. That is magic! When coders know exactly what needs to be done and tasks are broken into many smaller objectives, each is implemented and corroborated without slowing forward momentum. Many novice programmers write all their code and then start the lengthy process of debugging. This painful process is often drawn-out when logic and alien behaviors arise. This approach often takes considerably

DOI: 10.1201/9781003016427-3

longer than when writing one line at a time. When the coder is not blessed with magical fingers from which perfect code flows like a river, the tried and true method of writing one line of working code at a time often delivers the fastest results.

ONLINE INSTRUCTION

The internet is an outstanding informational resource. Almost anything the technical artist needs to know can be found on the web. This is a blessing and this is a curse. While any single piece of guidance can be found, there is often too much information, some of which is relevant and some less than credible. Some information is created with different target audiences in mind. Some is simply out of date. How is a novice technical artist programmer supposed to navigate through this endless sea of information?

The credibility of anything found on the web is dubious. There are many fine instructors as there are many videos created by individuals who have no idea what they are demonstrating. The only way for beginning programmers to understand the value of online instruction is attempting to follow the instructions, examining the quality of the results, and answering the following questions. Did the resource solve the problem? It is rare to find a web-based instruction that addresses the technical artists' immediate needs. Was the instructor able to communicate the solution effectively? Did she explain each step along the way or did she expect each step to be mimicked? Was the technique delivered in such a manner as to be understood and implementable? Understanding how to push buttons and pull sliders may solve immediate problems but does the information transfer to other situations?

Unless the online information is created by a technical artist for technical artists, the information may be challenging to understand. Many solutions are provided by traditionally trained programmers who provide solutions for other trained programmers. Their language as well as their foundation of understanding may be beyond the technical artist's experience level. While attempting to understand the context of the solution, technical artists can lose focus and never solve their original problem. How do novice programmers know when the presented solution is worth the time and energy to understand and implement?

Time is technical artists' most valuable resource. When presented with unknown problems or situations, she must find solutions in the shortest time possible. Understanding resource credibility requires time and patience. Technical artists must employ trials and errors to implement and test the results of new online resources. Is the solution's thought process transferred effectively? Can the technique be extrapolated for solving other problems? Does the resource present information in an understandable language? The time required to vet solution plausibility is often greater than the time required to generate an original solution. Is the solution a black box? Black boxes are dangerous chunks of blindly pre-generated code, modules, or plugins with no functional or implementational understanding. (Nightmares of generating fractal harmonics or edge detection come to mind.) Do not alter the structure of original scene files to conform to the needs of black boxes. Running untested processes without thorough understanding is dangerous. Unless necessary, and carefully tested, black-box solutions should be avoided.

EXTERNAL CODE REFERENCES

External code references are found everywhere. Websites, online tutorials, and books offer unlimited, pre-written examples. Code may even be borrowed or taken from other writers. With enough dedication and resourcefulness, technical artist programmers should be able to find pre-created code to do whatever is needed to accomplish their tasks. Found code has potential to solve relevant problems quickly without re-invention. They provide working demonstrations how techniques and algorithms are intended to be implemented. They even expose new perspectives by approaching problems and implementing solutions from different and unexpected angles.

As beneficial as the results may seem, as a good rule of thumb, technical artist programmers should never use externally generated code unless it is thoroughly understood. Running code without understanding is another type

of black box. While externally generated references have their benefits, they also have their disadvantages. They may force coders into a direction with disadvantageous directions from which there are undesirable or disastrous consequences. Ironically, implementing unfamiliar code often consumes more time than the promised savings. The time required to hunt down the sources of unfamiliar errors and unanticipated results often takes longer than when implemented by-hand. The code itself may be challenging to understand. It may be written poorly, use math beyond the understanding of the programmer, or use an exotic or unfamiliar language structure. Understanding these factors before implementation saves time and effort in the long run.

Technical artist programmers should do their best to reverse engineer every line of pre-generated code. However, when time is of the essence, this may not be a possibility. Who has the luxury of vetting all python library code? Hopefully, there is robust and easy to understand supporting documentation. When that is unavailable, the only recourse is thorough experimentation. Understanding output and how input impacts its behavior should eliminate most unexpected results. Regardless of documentation and experimentation, it is the technical artist's responsibility to understand any external code before implementation.

CONCLUSION

Most technical artists do not come from traditional software engineering backgrounds. This limitation should not prevent them from generating clean, safe, and functional code. Clearly understanding required tasks and solutions is essential for generating effective results. Before writing any code, automated solutions should be executed by-hand first, ensuring thorough understanding of essential steps. Coding is not magic, and the chance of getting it right in your first attempt is small. Exercise patience. Writing, testing, and understanding one line of code at a time result in fewer mistakes and faster solutions. Instructions, tutorials, and other guides found online need to be treated with care and diligence. They may be fantastic places to start when solving challenging situations. However, without thorough investigation and verification, these resources may create more trouble than they solve. The same diligence must be employed when implementing externally generated code. When possible, every line of code should be reverse engineered to verify understanding and correct results.

INTRODUCTION

Computer languages provide interfaces between technical artists and their CPUs and GPUs. While there are dozens of different programming languages, this chapter only deals with the languages which technical artists deal with on regular bases. Some languages, such as Lisp, are becoming less common, while others such as Rust are up and coming. This chapter only deals with the languages technical artists are likely to encounter at the time of writing.

Technical artists utilize whatever available tools are needed to deliver digital art. The languages listed in this chapter are ordered in terms of their absolute effectiveness and potential with respect to technical artist responsibilities. Avoiding political discussion on which languages are best, (an emotionally charged topic), the author wishes to declare that all the listed languages are excellent and very effective. Each has its best use cases and should be employed when the circumstances call for them.

Of the listed languages, some are compiled, some are interpreted, and others are hybrid. This chapter is written from the perspective technical artists work on Microsoft Windows or Apple macOS operating systems. The use of UNIX-based operating systems in the real-time community, at the time of writing, is scarce yet common in the off-line community. Since this book's primary focus is on real-time applications, its primary attention is on readily available, real-time languages.

C/C++

C and its extension C++ were developed in the early 70s and 80s and are two of the most influential languages. Most modern languages such as C#, Java, Java Script, and Python are C derivatives. C is the instrumental backbone of modern operating systems such as early versions of Unix and Linux. C++ was released in 1983 and has since become the primary language for most primary high-performance software such as Microsoft Office, Adobe products, and modern real-time engines such as Unreal and Unity. Within the computer graphics industry, most major digital content creation tools (DCCs), applications, and operating systems are written in C or C++.

C and C++ are sensitive and responsive languages. Like high-end sports cars, they trust that programmers know what they are doing. With understanding of what needs to be done combined with accomplishment strategies, a programmer can do just about anything. They are compiled languages and are translated to machine code which processors execute. This makes them fast and efficient and gives potential to deal directly with the CPU, access sensitive memory information, and control external devices. When necessary, programmers may extend the language to meet specific needs. Translating to machine code ultimately means that any kind of program can be created, from low level systems programming to high-level graphics user interfaces (GUIs).

On the opposite side of the spectrum, C and C++ can be hostile, unforgiving languages. C is a crude and primitive language with no safety mechanisms. Memory pointers are powerful tools and are essential components to both languages. Their utility demands much responsibility and could potentially crash programs and even machines, when used improperly. Neither language provides convenience functionality for memory management, and force programmers to do most of the work themselves. C++ is large, complex, and complicated and is a cognitive load for most programmers. Learning either requires learning language mechanics at the same time as problem solving, slowing the learning pace and requiring rigorous practice.

DOI: 10.1201/9781003016427-4

While C and C++ represent the most powerful and prominent languages for technical artists, most will rarely have the opportunity of using them. Artists who are members of high-end tools teams will program in C++. Because their explorations often go beyond the reach of commercially available DCCs, high-level research and development artists will most probably program in C++ as well. Other than these high-level tasks, technical artists rarely have opportunity for programming in these languages.

If C and C++ are hard to learn, challenging, and are rarely used, should technical artists learn these languages? The answer is a resonating, "Yes!" C and C++ are the language choices of the most experienced and competent technical artists. Most other languages at technical artists' disposal are derivatives. Once mastered, all new languages can literally be learned over night. Of course, thorough comprehension takes longer but functional competence is achieved in short time. Unbound by interface or DCC functionality, technical artists can create their own tools and applications. They can explore topic areas where no commercially available products have ventured. Most DCCs and applications provide C or C++ SDKs empowering technical artists to build onto and extend existing packages. Within the context of production environments, technical artists with C or C++ understanding provide valuable assistance debugging frustrating real-time engine issues. Technical artists are not necessarily better programmers than the engineers. But as artists, they have different perspectives on how art data structures are used and manipulated, and anticipate "creative" ways artists may use the data. Except for the time required to learn these languages, technical artists have everything to gain and nothing to lose through their mastery.

PYTHON

While not as fast or as capable as C and C++, all technical artists should have a working, functional understanding of Python. This book is devoted to the development of technical artist Python skills. Almost all DCCs and real-time engines now have Python interfaces. It is supported on Microsoft Windows, Linux/Unix, Apple macOS, and other operating systems as well. It is available as a permissive software license which is compatible with the GNU General Public License. Being permissive allows programmers to customize Python's libraries and distribute their alterations. It is readily available and easily accessible. Python is the primary scripting language for the computer graphics industry.

As a tribute to the British comedy group, Monty Python, Python was created to be easy and fun. Its design philosophy promotes code readability and empowers programmers to write a clear, logical code for small- and large-scale projects. Python is readable and well-structured which makes it easy to learn and is often used as an introductory language. It is dynamically-typed and garbage-collected, freeing programmers from memory management and the responsibility of declaring variable types before usage. Like C and C++, Python is object-oriented and functional which makes it as productive as the prior languages but requires less code to accomplish the same tasks. It is an interpreted language which stops execution when bugs are encountered and reports their locations and logical traceback which helps with debugging. Almost all major DCCs and real-time engines support Python or can be easily extended to support it. Python is also the language of choice for artificial intelligence (AI) and Machine Learning. While these fields have not yet integrated with real-time rendering, Python's availability will facilitate the combination. AI is already becoming a familiar component in animation, motion capture, and rigging.

While Python is a good alternative for C and C++, it does have its drawbacks. Because it is interpreted, it tends to be computationally "slower". The interpretation and dynamically-typed variables are computationally expensive and require extra memory. Some find Python to be too simple as it is harder to achieve fine machine control afforded by C and C++. With extra effort, it can be extended to these languages. While Python is interpreted and good for identifying syntax errors, it also tends to generate more run-time errors.

Python is the essential language for technical artists to learn. Since most of the major-packages and real-time engines support it, it is an excellent language for duplicating and automating artist workflows. Almost all pipeline work and project management tasks are performed in Python.

HLSL/GLSL

High-Level Shading Language (HLSL) and OpenGL Shading Language (GLSL) are the two primary languages used for programming real-time graphics pipelines. HLSL is a proprietary language developed by Microsoft for the Direct 3D 9 *Application Programming Interface* (*API*) and has a syntax based in the C programming language. GLSL is also a high-level shading language developed by the OpenGL Architecture Review Board (ARB) and shares the same C based syntax. The two languages are analogous. Apple created its own shading language called Metal Shading Language (MSL). MSL is used for Apple devices and is rare to see outside of these contexts.

Almost all commercial real-time rendering engines speak either HLSL, GLSL, or both. All real-time graphics editors and visual effects packages utilize either language. Traditionally, HLSL has been directed exclusively toward Microsoft devices and GLSL for everything else. HLSL should run faster on Microsoft products, while OpenGL should run faster on everything else. Up and coming platform, Vulcan, can take either language because of its SPIR-V intermediate language ecosystem.

It should be every technical artists' goal to attain at least a working familiarity with either of these languages. (A fundamental introduction to HLSL is included in *Lighting and Rendering for the Technical Artist*.) Most real-time rendering engines and packages support either or both. While not essential for entrance into the industry, careers are significantly embellished by their understanding. HLSL and GLSL are very similar to each other and the exercise of translating shaders between them not only increases language familiarity with their subtle nuances but also reinforces effective shader programming skills. Neither is necessarily better than the other and should be chosen based on the resulting end devices: HLSL for Microsoft and GLSL for everything else.

C#

C# was developed by Microsoft as part of the .NET framework, designed for the Common Language Infrastructure (CLI). It is a statically-typed compiled language which means that it should outperform other interpreted languages such as Python; yet its development time is slow due to the compiling overhead. Like Python, it is an object-oriented language making it ideal for generating procedural workflows. In 2017, Microsoft made C# open-source and available free of charge.

Within the context of real-time rendering, C# is the go-to language for Unity Engine development. Unity supports two scripting languages, C# and UnityScript (also known as Javascript for Unity). While the two languages are effective, all of the Unity libraries are built using C#. Unity engine considers C# to be the true canonical language for its development. C# is also used in the development for projects and applications for the Microsoft Hololens augmented reality device.

C# is a popular developer's language because of its design and object-oriented paradigm. Programmers familiar with Java are able to learn it quickly. However, the language has more constructions than Python such as assemblies, namespaces, classes, and methods which are essential to begin programming. Novice programmers may be more intimidated by C#'s structure compared to Python. While Python shares similar constructions, understanding is not required to begin programming.

C# is an outstanding language for supporting Unity and Microsoft integration which rely on standard syntax and libraries. It is essential for tools development within studios built upon the .NET framework. However, outside of the context of these specific situations, C# is not as popular or widely used as Python. These reasons should not

discourage the technical artist from learning C#. However, unless working in an environment where the language is essential, other languages such as Python should be considered first.

JAVA

Java is arguably one of the most popular of all computer languages and is used everywhere especially in client-server web applications. It is both a compiled and interpreted language because its source code is first compiled into a binary byte-code and then is usually run on software-based interpreters known as Java Virtual Machines. Released in 1995 by Sun Microsystems, Java was designed to have as few implementation dependencies as possible. It is the primary programming language for Android mobile game development and, when combined with Flash, used for other web-based games.

Java is simple, object-oriented, and platform independent. While memory is easier to manage than C and C++, it does require management. Because it is both a compiled and interpreted language, Java will run slower than compiled C and C++. Its "write once, use anywhere philosophy" is good for generating predictable behavior on multiple platforms but also lacks the specific device control C and C++ afford. While simple real-time rendering engines and games have been created using Java, creations of such applications are rare and usually performed as exercises or experiments.

OTHER SCRIPTING LANGUAGES

During the daily course of real-time development, technical artists will need to be familiar with a plethora of situational dependent languages. These languages are typically devoted to special use cases and local environments. While most of the included languages do not have common acceptance, technical artists should be familiar with them and be prepared to learn or re-learn them in little to no time.

JavaScript

After substantial understanding of one of the languages mentioned prior in this chapter, all technical artists should have cursory understanding of JavaScript. Commonly used for dynamic behavior and visual effects to web pages, JavaScript has many real-time applications. It is one of the three scripting languages used with Adobe Creative Suite. It, however, is the only language which can work on Microsoft Windows and Apple macOS operating systems.

Bourne Shell/C Shell

Bourne Shell and C Shell are popular command-line shell interpreters used for interfacing with Unix-like operating systems. Each is a command language, used for providing operating system commands, and a scripting language providing procedural system control through *shell scripts*. Bourne Shell, and its modern incarnation Bourne-Again Shell (bash) can be found on most Linux implementation. Bash is still available on Apple macOS systems while its default is now *Z Shell*(ZSH). C Shell and its improved version, *tcsh,* can be found on most traditional Unix implementations.

As combinations of command languages and scripting languages, shell interpreters are superior pipeline workflow tools. Novice users may be discouraged by the lack of graphics user interface (GUI). However, after an acclimation period, users will become aware that when used effectively, shell interpreters are faster and more powerful than any GUI. The ability to swiftly and easily customize the interfaces to technical artists' own styles make shells unparalleled production management tools. Not all production environments are built upon Unix-like operations systems. For those environments that are, technical artists must attain shell mastery in order to perform their responsibilities effectively.

DOS

Windows Batch Scripting and Windows *Powershell* are Microsoft Windows-based shell interfaces built on older *DOS* (Disk Operating System), commands, and logic. Like Unix-like shells, these interfaces are command languages and scripting languages. Based on archaic DOS, Batch Scripting is more limited and awkward than other shell languages. Powershell is Microsoft's suggested way of interfacing with the operating system. Created with enhanced logic control, it provides significantly improved control over traditional Batch Scripting. However, due to the condition of adding an extra module to the environment, Powershell may not be a viable alternative for many production environments. Due to DOS-based awkwardness, limitations to Powershell implementation, and the removal of the requirement to run Windows with DOS, most Windows users stick with the base GUI. While certain technical art tasks demand Batch Scripting or Powershell usage, most technical artists in Windows-based environments can perform their daily responsibilities without the mastery of these tools. (In fact, there are many practicing technical artists who are unaware of the Windows command-line or Powershell.)

HScript/VEX

HScript and VEX are the embedded languages inside Side FX Houdini. HScript is the legacy scripting language present since the first version of the software. A bit more like C Shell, the language is more than robust enough to replace the interface and expose full control to the user. It is common Houdini usage to operate the software without the graphical interface through pure command-line HScript usage. While the Python API is the suggested scripting language, HScript is still present to ensure backward compatibility. Users also find HScript's brevity more convenient and less verbose than the Python equivalent.

VEX is a small, efficient, and general-purpose language used for writing in-software shaders and custom nodes. Roughly based on the C language, VEX is inspired by the Renderman Shading Language to be a fast language for modifying digital assets. Due to its SIMD (Single Instruction Multiple Data) architecture, all points of data are processed in parallel, making VEX faster than HScript or Python and almost as fast as compiled C++. However, this structure requires concurrent data access and more elaborate algorithms, preventing it from being an all-purpose language.

MEL

MEL, (Maya Embedded Language), is Autodesk Maya's metaprogramming language which instructs its node architecture to perform tasks and solve problems. In the software's early versions, Maya and MEL were the same. Any performed operation originated from a MEL command. The language is outstanding for speeding up complicated or repetitive tasks. Written to feel a bit like the PERL and TCL languages, MEL can be somewhat limited as it lacks associative arrays and object-orientation. However, as an interface processing language, it performs its purpose very well. These days, Maya has a Python API with a Python interface to both MEL and to the C++ API, and MEL is mostly supported as a legacy language.

MAXScript

MAXScript is the built-in, Autodesk 3ds Max scripting language. 3ds Max is a very popular three-dimensional modeling and animation software and is considered to be easier than Autodesk Maya. It is limited only to Microsoft Windows. MAXScript can be used to automate repetitive tasks, combine existing functionality, and create new tools and interfaces to speed up existing workflows. The language has a loyal support base. While 3ds Max may also support Python and C# APIs, MAXScript is the go-to language with the widest functionality.

Lua

While not as prevalent as the prior mentioned languages, *Lua* is a popular development scripting language technical artists will encounter. It is a lightweight language used in game programming or as an embedded scripting tool. It has a similar structure to Python but is simpler to learn and smaller to embed.

CONCLUSION

There are many computer languages available to technical artists but only a handful will be practical on an everyday basis. If there could only be one language to learn, all technical artists should have a fundamental understanding of Python. Nearly every digital content creation (DCC) software and real-time engine support a Python interface. Python is the language of real-time, three-dimensional computer graphics, as well as artificial intelligence, machine learning and many other disciplines. It often behaves as the glue between different applications. It was designed to be easy to read and easy to learn. As a modular, object-oriented language, it grants users considerable capability with only a small amount of code. It is a slower language than many others. However, its functionality outweighs its handicaps.

When presented with the opportunity, all technical artists should learn C or C++. Nearly all DCCs and real-time engines are written in C++, or have C++ APIs for writing high-performance tools and plugins. They are the Latin of modern computer languages. Mastery will make the learning of other languages a simple task. Regretfully, students must devote months of study and practice to master the languages.

Other languages are excellent when dealing with specific technical artist tasks. C# is an excellent, all-around language and is essential for working within the Unity real-time engine. HLSL and GLSL are crucial languages for interfacing and editing real-time rendering pipelines. Most DCCs have built-in languages which maximize tool control. HScript and VEX are used in SideFX Houdini. MEL is found in Autodesk Maya. MAXScript is used in Autodesk 3dsMax. There are multiple shell scripting languages which are very helpful for operating system level pipeline management. Bourne Shell and C Shell are used in Unix-like systems. DOS is used in Microsoft Windows.

INTRODUCTION

Inspiring artists to program can be challenging. Overcoming traditional social biases through the inception of greater creative freedom is difficult to impart. Becoming a great programmer is a life-long pursuit. However, fundamental and relevant Python can be taught to artists in a six-week curriculum. The class is by no means thorough or encompassing. However, it introduces the students to the programming world in a manner that is relevant to them using familiar 3D tools. Fledgling technical artist programmers do not need to be experts. They need to create code that is clean, safe, and functional which helps encourage creative workflow. Working with familiar tools and solving familiar problems provide opportunity for artists to start their programming journey. This book utilizes the SideFX Houdini DCC as the Python integrated development environment, (IDE). Not all students will be familiar with Houdini but will be familiar with traditional shapes, objects, and materials regardless of their background. Some may argue the effectiveness of using a commercial interface as a Python IDE instead of using the command-line. While it is true the command-line approach is more powerful, the goal at this stage is to encourage artist potential which is easily achieved through friendly, familiar environments.

CHALLENGE OF TECHNOLOGY

Ironically, technological development may not encourage artist programming but may unintentionally hinder it. Due to interface limitations, software developers bloat their products with greater amounts of functionality. The internet provides instruction and guidance for employing this functionality. However, there is too much information on the internet and navigation through these informational mazes is challenging. As toolsets increase in functionality, artists are expected to produce more content in less time. The ever-increasing churn for content does not promote time for developing career enhancing skills.

TRADITIONAL TRAINING

Python taught in traditional, structured environments will be effective and thorough. However, this thoroughness and lack of relevant applications fail to inspire artist interest. Traditional courses need to provide the greatest amount of material for serving the greatest number of potential applications. For technical artists, there is need for concise instruction which quickly instructs only relevant skills in the most rapid timeframe.

PYTHON IN SIX WEEKS

In the university program where the author instructs, students have only 12 months to learn everything they need for embarking on careers in the game, film, or simulation industries. Removing vacations, breaks, time in-between semesters, and other required classes, technical art students have only six weeks to prepare to support their 20-plus person capstone project teams. Once their collaborative team projects start, the students' ability to focus on intensive, essential skills for supporting their teams is diminished. The six-week period is all they have to cement strong Python programming foundations until they decide to dedicate future effort once attaining employment in the industry. This period must not only instill a solid programming base but must also demonstrate, through example, how they can immediately put their skills to work supporting their teams. The instructional curriculum must cover only the essentials and demonstrate using only familiar, pragmatic situations found in everyday real-time development.

DOI: 10.1201/9781003016427-5

The details of this curriculum are described in this book. The course topics include the following:

1. Python setup within a familiar commercial DCC, (Houdini)
2. Python basics including printing, comments, and simple expressions
3. Python logic structure
4. Python functions
5. Intrinsic Python data types
6. Object-Oriented Programming, including inheritance
7. Python exceptions
8. Python file processing
9. Python graphical user interfaces (GUI)
10. Standard Python coding standards

Is this curriculum thorough? No. The covered topics barely give students enough resources for reverse engineering some external code and writing their own simple scripts. To obtain more thorough programming mastery, students may need to spend considerable time studying additional resources or even re-take a more formal course. When presented a second time, the information will be absorbed more efficiently and cleanly.

Is the curriculum adequate? Yes. In the short-term, students will have the fundamental tools for performing guerilla-like programming support for pipelines, rigging, and artist support. Technical art students will be able to immediately support their teams to whatever capacity they are willing to try. By consistent application of their learned topics, they will be able to cement solid programming skills which will further reinforce and expand with continual use. Students will have only enough Python understanding to accomplish basic technical artist responsibilities yet have enough room for continuous learning.

Is the curriculum effective? The answer to this question is dependent on the students and their willingness to apply what they have learned to their daily operations. Regardless of their skill, the students who proactively apply themselves will advance more rapidly than those who program only when requested or are required.

FAMILIAR ENVIRONMENT

Working on familiar and relevant real-time development responsibilities is the primary benefit for learning Python in a 3D environment, such as Houdini. Beyond file organization and manipulation, technical artist programming focuses on three other items: scene objects, polygons, and colors. Scene objects may be environments, characters, props, visual effects, sounds, lights, cameras, or other supporting constructs such as regional volumes. Polygons are the object composing components. Polygonal data may include points, vertices, lines, polygons, and adjacencies. Colors are the primary properties defining the visual appearance of these objects in real-time rendering engines.

The examples in this Python course focus on tasks such as scene object creation, position, orientation, the manipulation, manipulation of colors, and pure polygonal generation. Introducing Python concepts to address these tasks encourages artists to apply Python scripting to everyday and future situations.

Real-time application APIs can be challenging to learn. Conveniently, technical artists are already familiar with their results. Additional API functionality instruction is not required. All relevant API code is provided with minimal explanation to encourage the focus on Python language basics, not on the DCC and its API.

HOUDINI IDE

SideFX Houdini provides the Python environment used in this text. A free learning version, *Apprentice* , is downloadable from the SideFX website, https://www.sidefx.com. Installation instructions are provided in Chapter 6 of this book. This free learning version of Houdini is limited and not adequate for full CGI production but fully supports Python and is functional enough to provide an ideal Python learning environment.

There are many reasons why Houdini is selected as the Python development environment. Houdini is supported on Microsoft Windows, Mac OS, and Linux. The Python language exposes the object-oriented nature of the API and lends itself for what can be described as "Pythonic" code writing. The package provides multiple Python user interfaces ranging from the trivial to the complex. The easiest is demonstrated in this book. The language implementation is stable. The evolution from Python 2 to Python 3 has had only a minimal impact on the language implementation. Any attribute alterable through the graphical interface is easily adjusted through the object-oriented Python API. Houdini is an extensive, package and instruction on the full Python API would be exhausting. This course focuses only on teaching fundamental, technical artist Python and exposes only enough Houdini to provide support.

This Python course instructs fundamental Python 3. While Houdini has supported Python 3 for many beta versions, the first production version using Python 3 is Houdini 18.5.

Could this Python course be structured with other DCCs such as Autodesk Maya, Blender or Autodesk 3DS Max? Yes! These other packages are professional staples and the course could easily be adjusted to collaborate with any of them. As of the time of this book's writing, the author is most familiar with Houdini and its Python implementation. Please contact the author for assistance in integrating the course with any of these packages.

The author does not suggest attempting to integrate the course with modern real-time engines such as Epic Games Unreal, Unity, or Nvidea Omniverse. While these engines do support Python integration, their implementation is simply too recent to be understood or too complicated to introduce Python basics. This book addresses the basics of the language first. Once understood, later integration into these environments will be easier.

COMMAND-LINE VERSUS IDE

Students may consider working on the old-school style of programming in the command-line instead of within a DCC IDE. Working in the command-line means using external editors to write and edit code and use systems' command-line interfaces to execute scripts. PowerShell and the cmd.exe command prompt are the operating system tools for Microsoft Windows. The *Shell* Prompt is the tool for Linux and Mac systems.

There are many reasons why working from the command-line is an advantageous way of learning. Zed Shaw from the *Learning to Program the Hard Way* series believes very strongly in this approach and the author of this book agrees. The command-line is the primitive computer interface and programming this way teaches direct interaction. Once students understand how computers function, programming tasks become less intimidating. Assembling sequences of instructions generating specific behavior becomes obvious and less mysterious. Working from the command-line is more effective than from graphical user interfaces, especially when supporting production pipelines.

While this book aims to instruct technical art students with only the essential Python required to launch careers in the game, film, and simulation industries, it also appreciates students who are less enthusiastic about learning program. Using DCCs exclusively is less intimidating than working from the command-line. Most modern DCCs have command-line access to their Python APIs. This book attempts to expose programming as simplistically as possible to beginning technical art students. Command-line access and functionality are available to more proactive students.

CONCLUSION

This chapter attempts to illustrate many of the mental issues facing beginning programming students. It exposes some of the insecurities and educational demands modern students deal with when learning new skills. Most traditional programming instruction is effective providing students with robust language understanding and covering any potential application. However, these courses are lengthy and provide extra material not needed for technical art students.

This book proposes to resolve this situation with a six-week introduction to Python programming directly focused on skills required by beginning technical artists. The course starts with a simplistic introduction to using Python within the context of a commercial DCC, basic print statements and simple expressions. The course then covers the Python logic structure, Python functions, and intrinsic Python data types. Considerable time is devoted to the introduction of object-oriented programming. Cleaner programming is reinforced with Python exception handling and file handling. The course concludes with the introduction of basic Python graphical user interfaces.

Introducing Python material within familiar production environments is effective with artists. They are familiar with scene objects, polygons, colors and their creation, modifications, positions, and orientations. SideFX Houdini is the DCC employed in this text. Houdini has built-in Python interfaces and its object-oriented structure integrates naturally with the software's API. Effort is made to focus on the basic Python structure instead of explaining more complicated Houdini Python integration. This book provides a Python course using the Houdini interface but could be adjusted to work with other DCCs such as Autodesk Maya, Blender, and Autodesk 3D Studio Max.

Instead of learning Python in the context of a commercial DCC, it is possible to learn through the basic command-line interface. There are many advantages of learning how to program this way as it teaches the students to think more like the computer does. However, this course provides artists with the easiest, most direct method for instructing the language as rapidly as possible. The command-line interface is left for more proactive students.

INTRODUCTION

This Python course begins with downloading and installing SideFX Houdini software. Depending on environment, this may have already been done for the students.

Once the software is installed and operating, students are given a quick introduction to the Houdini interface. The Python shell is introduced. This tool provides students their first exposure to Python by executing simple commands and performing simple mathematical operations. The concept of the programming variable is introduced. This chapter concludes with transitioning from the Python shell to Python script creation.

SOFTWARE INSTALLATION

Students who already have access to Houdini may ignore this installation section and progress to the *Python Shell* section of the chapter. The SideFX Houdini software package can be downloaded from the SideFX website, https://www.sidefx.com/.

The first production version of Houdini to support Python 3 by default is version Houdini 18.5. When downloading, please download version Houdini 18.5 or more recent.

SideFX requires an account to download the software. The account creation is accessed through the *Login* button in the upper right corner of the SideFX website. Figure 6.1 shows the *Login* button location.

The *Login* process requires a username and password. Since there is no account yet, the process is started by pressing the *Please Register* text. Figure 6.2 shows the *Please Register* location.

All the fields of the registration form designated with an asterisk, *, are required to be filled out. The *Terms of use* and *Privacy Policy* opt-ins are required but the subscription and profile visibility choices are optional. Finish registration by pressing the *Register* button. Figure 6.3 shows a completed registration form.

The software is downloaded from the *Get* tab at the top of the site. Choose the *Download* option. Figure 6.4 shows the *Download* option location.

Press the *Download Launcher* button to initiate the download process. Figure 6.5 displays the SideFX launcher window.

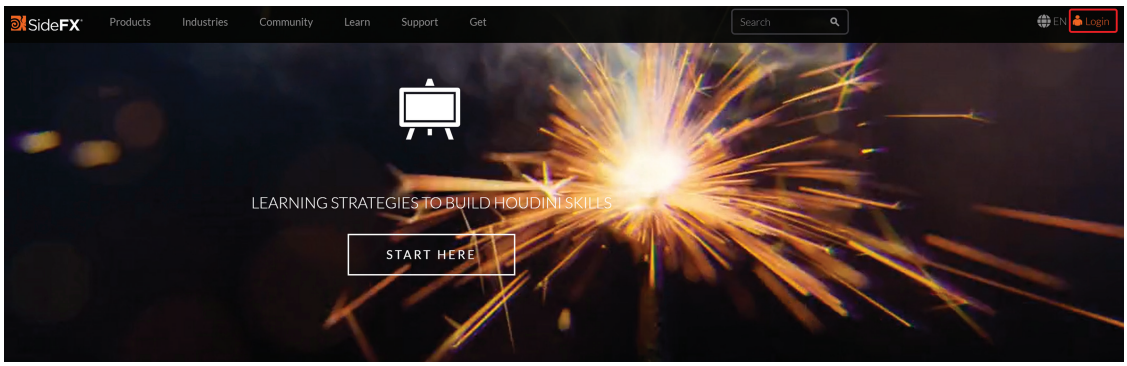

FIGURE 6.1 SideFX Houdini website.

DOI: 10.1201/9781003016427-6

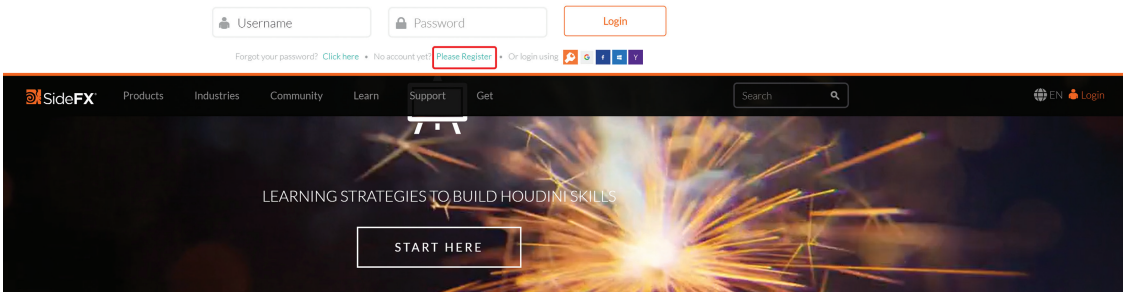

FIGURE 6.2 SideFX account login.

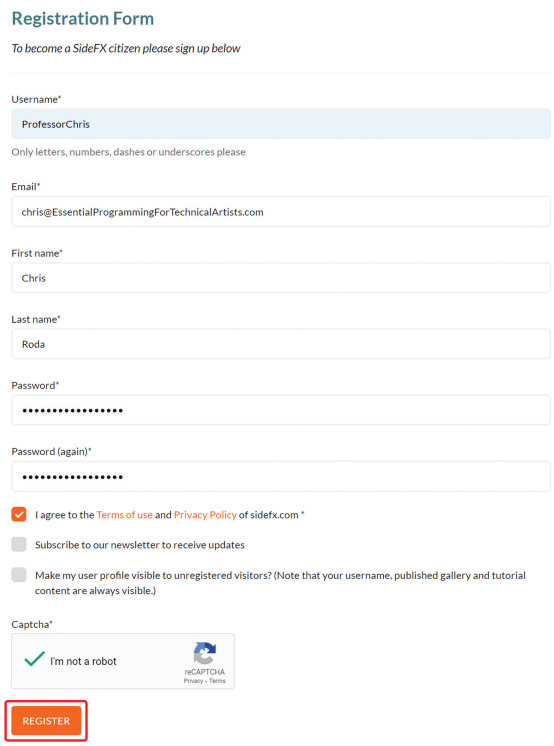

FIGURE 6.3 SideFX registration form.

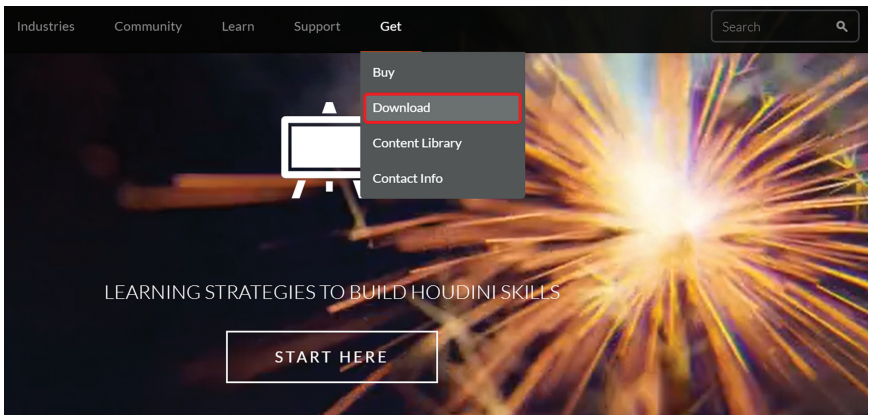

FIGURE 6.4 SideFX download selection.

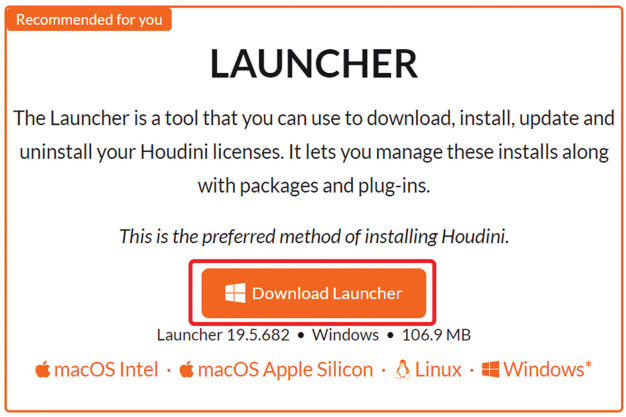

FIGURE 6.5 SideFX launcher option.

The *install-houdini-launcher.exe* should begin downloading immediately and will be stored in your *Downloads* folder.

Double clicking the downloaded file initiates the installation process. A setup window introduces users to the installation window. Press the *Next* button to continue. Figure 6.6 displays the Houdini setup window and *Next* button location.

The launcher will query for an install location. Unless there is a different desired install location, leave the default destination folder and press the *Install* button. Figure 6.7 shows the *Install* button.

The Houdini Installer should download and install. Upon completion, users are queried if the *Start Menu* and *Desktop* shortcuts are desired. Unless unwanted, leave the default settings and press the *Finish* button. Figure 6.8 shows the shortcut options.

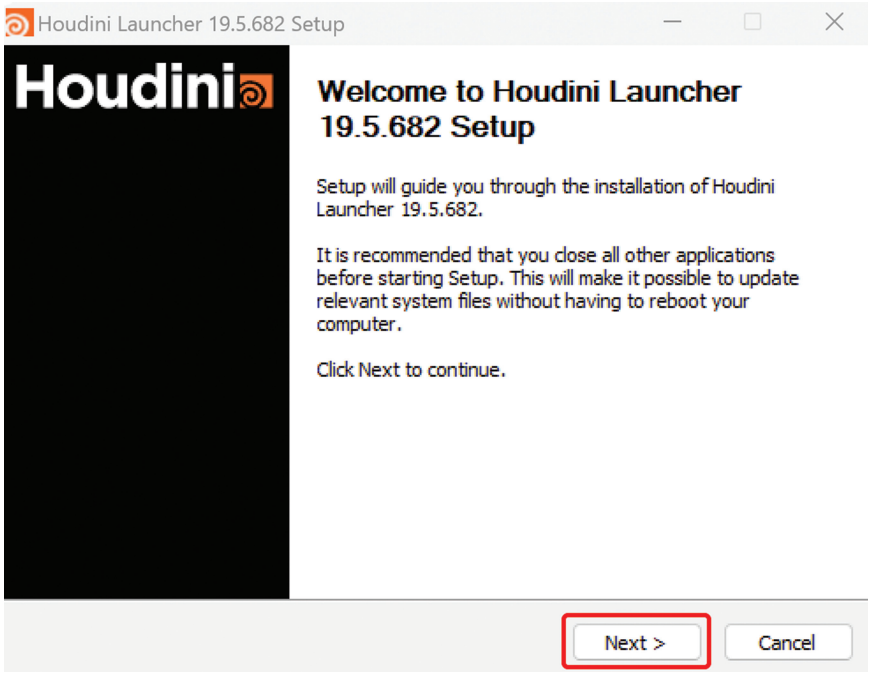

FIGURE 6.6 Houdini setup window.

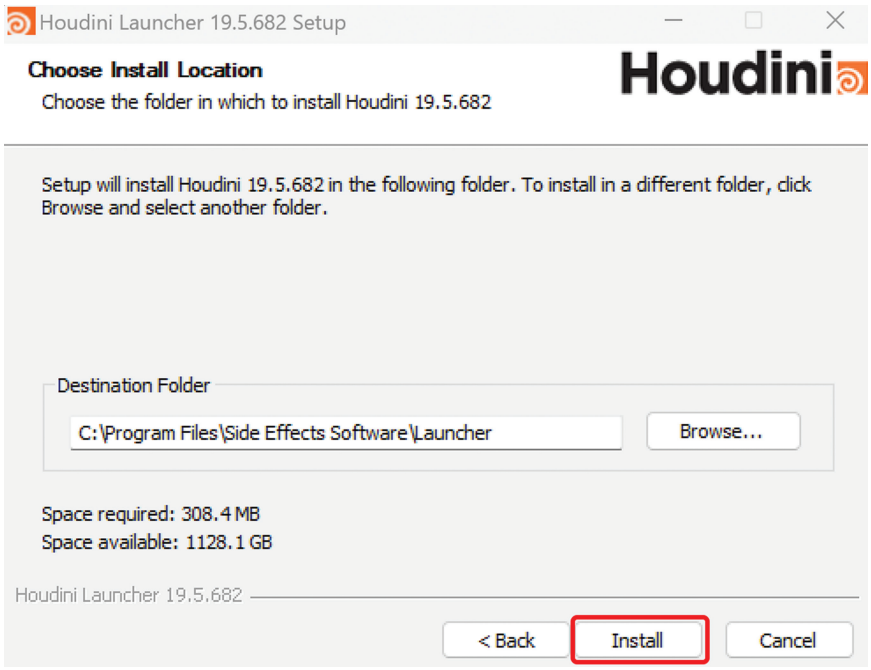

FIGURE 6.7 Houdini installer location folder.

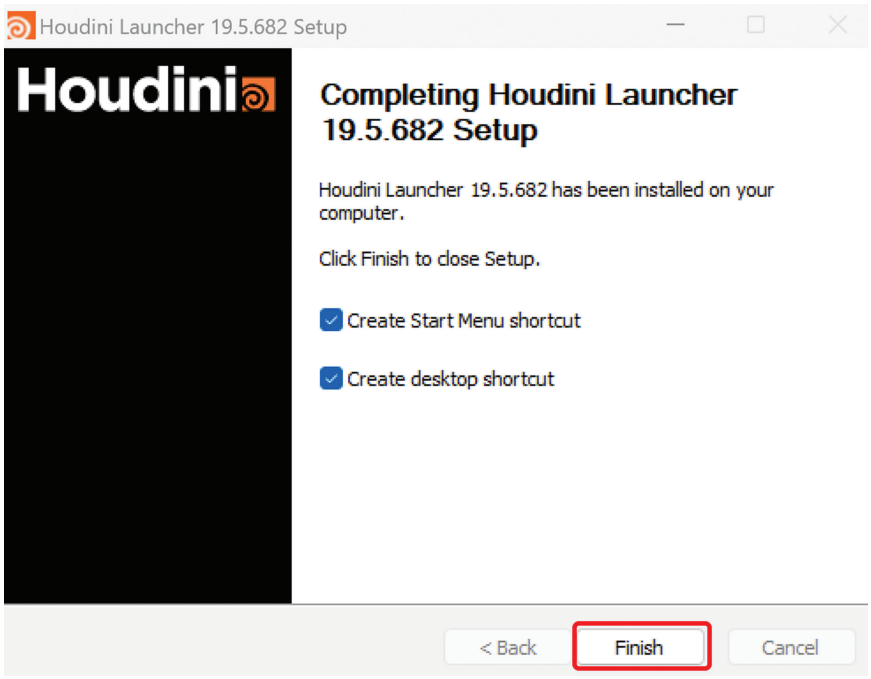

FIGURE 6.8 Houdini shortcuts menu.

FIGURE 6.9 Houdini launcher icon.

The *Houdini Launcher* icon should be installed on your desktop display.

Launch the *Houdini Launcher* (double mouse click). Figure 6.9 displays the *Houdini Launcher* application icon.

Prior Houdini Installation

If readers have never installed Houdini before, skip this section and go to the *Initial Houdini Installation* section. When readers already have a prior version of Houdini installed, they are queried if they want to import the Houdini Installations. Press the *Yes* button. Figure 6.10 chooses to import Houdini installations.

Select the *OK* button once all of the prior installations have been accounted for. Readers may now skip to the *Houdini Environment Setup* section.

Initial Houdini Installation

When installing Houdini for the first time, The *Houdini Launcher* window will appear. Press the *Log in* button to login to the Launcher. Figure 6.11 shows the *Houdini Launcher Login* button.

The user will be directed to the *Houdini Launcher* website. Enter the name and password used to register and hit the *Login* button to complete the process. Figure 6.12 displays the login process.

The user may now return back to the *Houdini Launcher* app window. Press the *Install Houdini+* button to select the Houdini version. Figure 6.13 displays the installation command button.

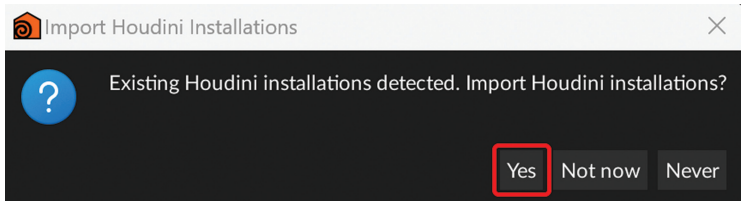

FIGURE 6.10 Houdini prior installation.

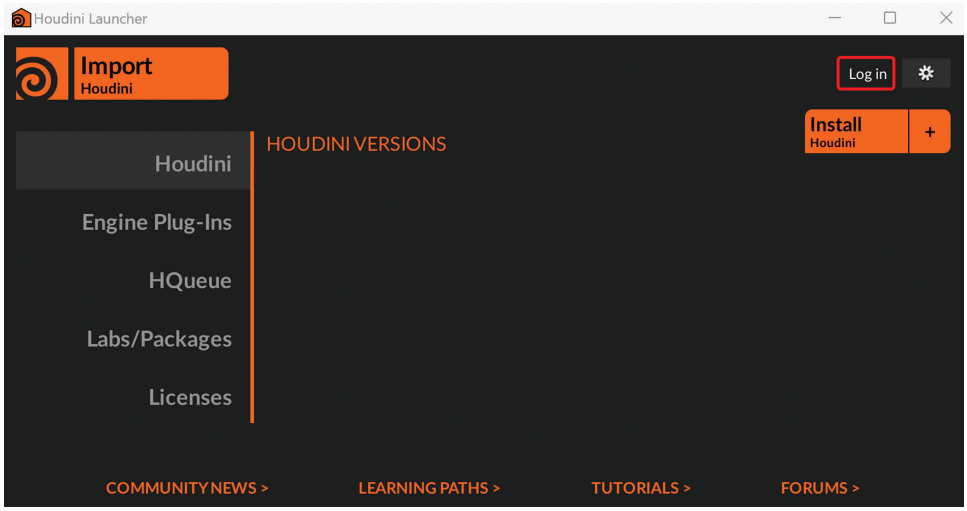

FIGURE 6.11 Houdini launcher.

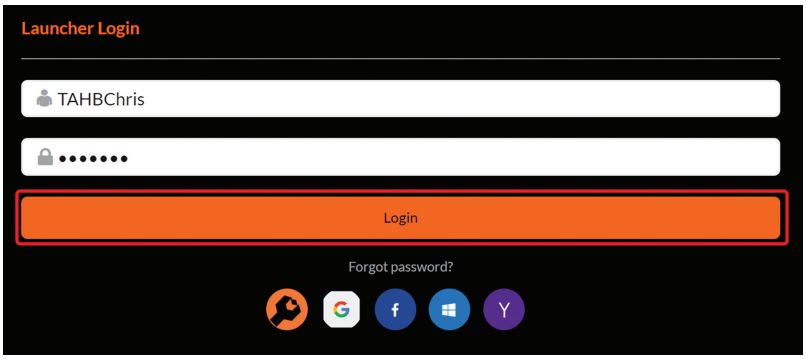

FIGURE 6.12 Houdini launcher login.

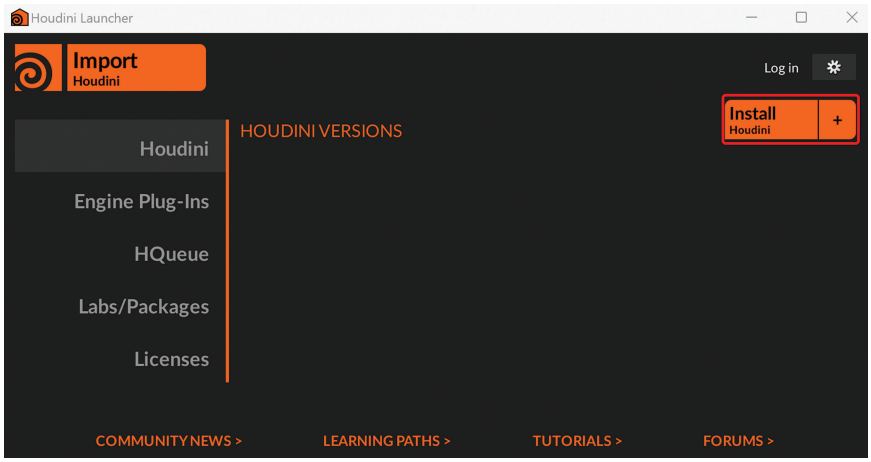

FIGURE 6.13 Houdini install +.

Select the highest number Production Build available in the *Version Selection* window. Unless essential, always select *Production Builds*, as in Figure 6.14.

In the *Houdini Preferences* window, the *Automatically Install License Server, Start Menu Shortcut, and File Associations* option should be selected. The *Create Desktop Shortcut* choice is optional. Press the *Install* button to continue. Figure 6.15 displays typical, default preferences.

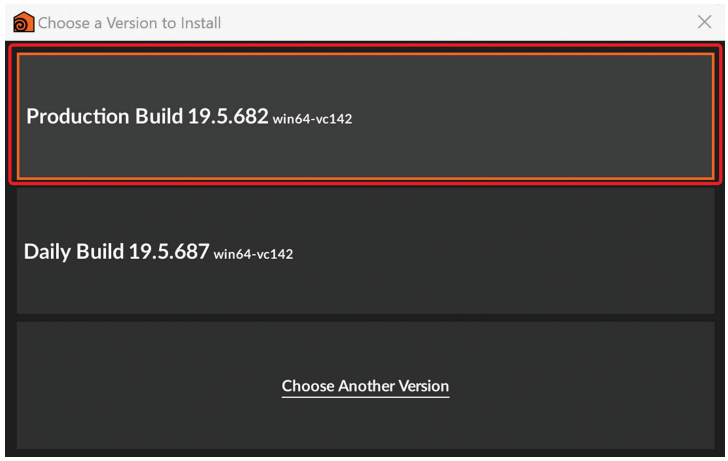

FIGURE 6.14 Houdini version selection window.

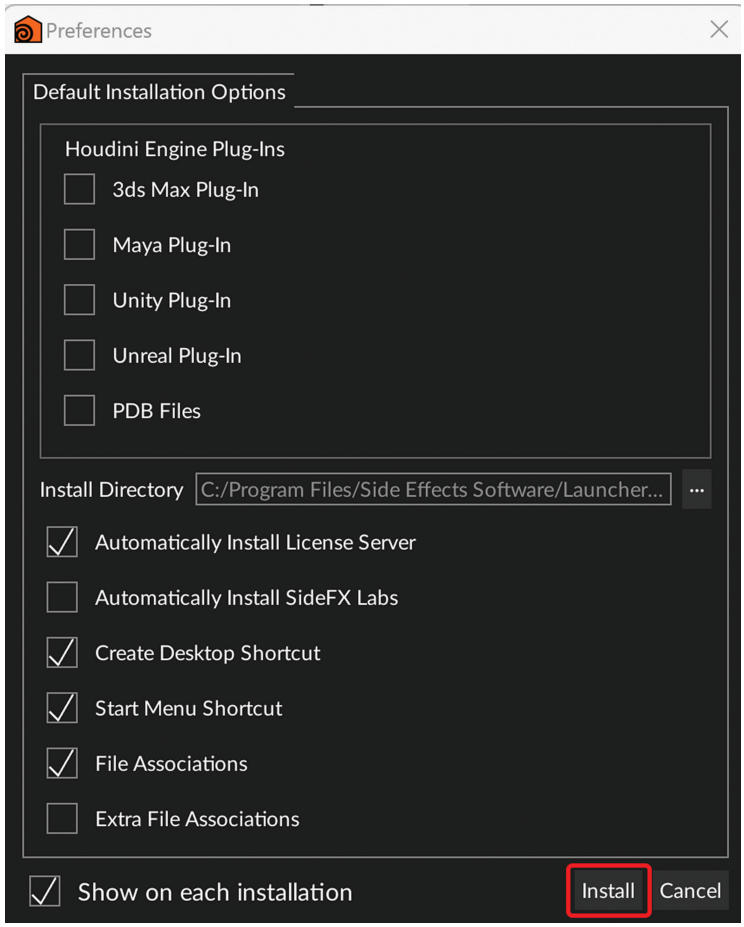

FIGURE 6.15 Houdini installation preferences.