

THE MONSTER FUN BOOK OF PATTERNS FOR DEVELOPING VIRTUAL REALITY APPLICATIONS

WITH EXAMPLES USING UNITY SOFTWARE

SHAUN BANGAY

The Monster Fun Book of Patterns for Developing Virtual Reality Applications
With examples using Unity software
Shaun Bangay

Copyright ©2019 Deakin University, Geelong, Australia

These materials are not sponsored by or affiliated with Unity Technologies or its affiliates. “Unity” is a trademark or registered trademark of Unity Technologies or its affiliates in the U.S. and elsewhere.

ISBN: 9781729142400

Contents

1	The Virtual Reality Pattern	11
1.1	Virtual Reality Applications	11
1.1.1	Description	11
1.1.2	Pattern	13
1.2	Manipulation	15
1.2.1	Description	15
1.2.2	Pattern	15
1.3	Locomotion	17
1.3.1	Description	17
1.3.2	Pattern	17
2	Representing and Manipulating Worlds	21
2.1	Scene Graph: A Pattern for Representing Virtual Worlds	22
2.1.1	Description	22
2.1.2	Pattern	22
2.1.3	Example	24
2.2	Maintaining Objectivity	25
2.2.1	Description	25
2.2.2	Pattern	28
2.2.3	Example	31
2.3	Adding objects to the virtual world	33
2.3.1	Description	33
2.3.2	Pattern	35
2.3.3	Example	35
2.4	Vectors: Representing Position and Scale	38
2.4.1	Description	38
2.4.2	Pattern	38

2.4.3	Example	40
2.5	Quaternions: Representing Orientation	43
2.5.1	Description	43
2.5.2	Pattern	45
2.5.3	Example	45
2.6	Where is everybody? Transformations, cameras and coordinate systems	47
2.6.1	Description	47
2.6.2	Pattern	51
2.6.3	Example	51
2.7	The Game Loop: Frame by Frame	59
2.7.1	Description	59
2.7.2	Pattern	59
2.7.3	Example	60
2.8	Having children	63
2.8.1	Description	63
2.8.2	Pattern	64
2.8.3	Example	66
2.9	Who's the boss: Managing Managers	69
2.9.1	Description	69
2.9.2	Pattern	69
2.9.3	Example	71
2.10	Exercises	75
3	Algorithmic Twiddles	79
3.1	All The Programming Skills You Will Ever Need	79
3.2	Variable Assignment	80
3.2.1	Description	80
3.2.2	Pattern	81
3.2.3	Example	82
3.3	Conditionals	90
3.3.1	Description	90
3.3.2	Pattern	91
3.3.3	Example	93
3.4	Iteration	95
3.4.1	Description	95
3.4.2	Pattern	96

3.4.3 Example	97
3.5 Abstraction	104
3.5.1 Description	104
3.5.2 Pattern	105
3.5.3 Example	106
3.6 Collections	115
3.6.1 Description	115
3.6.2 Pattern	120
3.6.3 Example	123
3.7 Fading with distance: Visibility, Scope and Life-time	126
3.7.1 Description	126
3.7.2 Pattern	128
3.7.3 Example	129
3.8 Recursion or I've seen this somewhere before	131
3.8.1 Description	131
3.8.2 Pattern	134
3.8.3 Example	135
4 What Could Go Wrong?	141
4.1 Debugging	141
4.2 Printing	145
4.2.1 Description	145
4.2.2 Pattern	146
4.2.3 Example	146
4.3 Watching in slow motion	150
4.3.1 Description	150
4.3.2 Example	152
4.4 Reductio ad collapsum	157
4.4.1 Description	157
4.4.2 Pattern	159
4.5 Testing and Unit tests	161
4.5.1 Description	161
4.5.2 Pattern	162
4.5.3 Example	163
5 Responsive Worlds	169
5.1 Event Handler	169
5.1.1 Description	169

5.1.2	Pattern	170
5.1.3	Example	171
5.2	Keeping track of time	173
5.2.1	Description	173
5.2.2	Pattern	174
5.2.3	Example	177
5.3	Taking control	179
5.3.1	Description	179
5.3.2	Pattern	180
5.3.3	Example	183
5.4	Locomotion in a particular direction	188
5.4.1	Description	188
5.4.2	Pattern	189
5.4.3	Example	190
5.5	Point and select	194
5.5.1	Description	194
5.5.2	Pattern	197
5.5.3	Example	197
5.6	Teleportation	198
5.6.1	Description	198
5.6.2	Pattern	200
5.6.3	Example	201
6	A Working Virtual World	205
6.1	State machine	205
6.1.1	Description	205
6.1.2	Pattern	206
6.1.3	Example	209
6.2	Autonomous agents	220
6.2.1	Description	220
6.2.2	Pattern	221
6.2.3	Example	225
6.3	Variety and unpredictability	237
6.3.1	Description	237
6.3.2	Pattern	238
6.3.3	Example	241
7	Physical in Virtual	247
7.1	Collision detection	247

7.1.1	Description	247
7.1.2	Pattern	248
7.1.3	Example	250
7.2	Forward Kinematics and Dynamics	254
7.2.1	Description	254
7.2.2	Pattern	255
7.2.3	Example	256
7.3	Inverse Kinematics	259
7.3.1	Description	259
7.3.2	Pattern	259
7.3.3	Example	260
8	All Together	265
8.1	Multiple users in multi-user virtual worlds	265
8.1.1	Description	265
8.1.2	Pattern	269
8.1.3	Example	270
8.2	Network Spawn	276
8.2.1	Description	276
8.2.2	Pattern	276
8.2.3	Example	277
8.3	Shared objects in multi-user virtual worlds	281
8.3.1	Description	281
8.3.2	Pattern	283
8.3.3	Example	284
9	Dynamic Geometry	293
9.1	Procedural Content Creation	293
9.1.1	Description	293
9.1.2	Pattern	295
9.1.3	Example	297
9.2	Dynamic Patterns	299
9.2.1	Description	299
9.2.2	Pattern	309
9.2.3	Example	312
9.3	Shaping up	316
9.3.1	Description	316
9.3.2	Pattern	318
9.3.3	Example	322

10	Shades of Grey	329
10.1	Lighting and Texturing	329
10.1.1	Description	329
10.1.2	Pattern	331
10.1.3	Example	334
10.2	Bump mapping	340
10.2.1	Description	340
10.2.2	Pattern	341
10.2.3	Example	343
10.3	Combining values	347
10.3.1	Description	347
10.3.2	Pattern	348
10.3.3	Example	349
11	Finishing Touches	353
11.1	Getting started	353
11.1.1	Description	353
11.1.2	Pattern	355
11.1.3	Example	357
12	Resources	365
13	Bibliography	369

About This Book

This book is about programming patterns used in creating virtual realities. There are a few conventions that are used throughout this book:

- Virtual reality (VR) is used as the generic term for the complete spectrum of virtual reality related contexts. This include the conventional head-mounted display based virtual reality system, but also desktop VR, augmented reality, mixed reality, extended reality, telepresence, pervasive reality, and so on. While many of these may have their own particular way of doing things, the patterns covered are usually applicable in all of these cases.
- Patterns are ways of abstracting the specific details of any particular virtual reality system so that we can easily recognize common structures that are used over and over again. This allows reuse of VR development expertise and encourages developers to concentrate on developing the unique and exciting elements of their applications rather than wasting time redesigning and re-implementing variations of structures they have created previously.
- The focus is on VR *application* development. An application is a specific virtual environment created for a particular purpose; such as a training world to teach particular medical scenario, a particular VR based game or an interactive visualization of a given data set. Applications are typically built using the facilities of a particular virtual reality engine. As such we assume the availability of the facilities provided by an engine which already provides graphical scene rendering capabilities, interface with specialist VR hardware, and include a way of representing all the information required to make up a virtual world. Building such engines is a separate topic, and well covered by existing texts on patterns in game development [Nystrom,

2014].

- This book does not assume any prior software development experience, but a programming background would be an advantage. Those lacking such a background are advised to utilize this text as part of a course module covering VR development and to take advantage of opportunities to interact with your tutors and teachers to develop the thinking and reasoning patterns required for programming. The content is aimed at a graduate or senior undergraduate university student.
- This book is not about documenting or using one particular VR engine. However we provide examples of how these patterns are applied in particular engines. These are presented separately from the actual patterns themselves to emphasize that choice of pattern is related to the application you are designing, and that the pattern can then be translated almost automatically into operational code on any appropriate VR platform. See section 11.1 on page 353 for details of a pattern with examples showing how to get used to working with a particular virtual reality engine.

1

The Virtual Reality Pattern

1.1 Virtual Reality Applications

1.1.1 Description

A virtual reality application incorporates most of the following components:

- The user consisting of:
 - a camera component representing the viewpoint of the user within the virtual world. This might logically contain two sub-cameras when stereoscopic viewing is involved.
 - various user input elements associated with that particular user. These components interface with the physical controllers and return information about how that controller is being used.
 - interaction controllers which translate input from the input elements into appropriate interaction; typically locomotion control for moving the user, manipulation tools for modifying the environment and interacting with other objects, and user interface elements for configuration changes.
- A scene containing all the objects in the virtual world, including the user, objects in the world and imposters for objects and user avatars being managed remotely.

- Managers of a multi-user virtual world distributed over a network of participating computers. This includes mechanisms for making sure that updates to the scene are made consistently across each device. Imposters are representations of other users, and shared objects, in the virtual world that may be connecting remotely over the network from their own copy of the VR application.
- Objects can be static (completely inactive) but are likely either be passive (responsive to external events), or active and undertaking autonomous behaviour specific to the virtual reality scenario and capable of generating actions which act on other elements in the virtual world.
- The objects in the scene are also subject to simulations that take into account their geometry and physical properties through processes that check for collisions, and simulate physics.

In addition to these standard elements, specialist virtual reality applications may include components to:

- Dynamically create and manipulate content in the virtual world. For example, this might include a component to generate trees to create a forest environment, or buildings to populate a city.
- Adapt the appearance of objects in the virtual world to accurately visualize information. Typically this would involve modifying the surface shaders on objects to enhance their appearance, the visual realism of the environment, or to display spatial information visually.

These components consist of objects (and their associated information) that appear in the scene but also actions that these objects need to take. Design of a virtual reality application needs to consider both information and actions to ensure that the application supports the desired interactions.

Common forms of interaction typical to all virtual reality applications are manipulation and locomotion. Manipulation involves modification of properties of an object by the user and typically mediated through a controller and the controller's virtual representation. Locomotion is the process whereby the user is able to navigate around the environment and visit different areas. There are a number of established conventions for achieving manipulation and locomotion but this does not preclude designing and developing new strategies to suit new applications and interaction devices.

1.1.2 *Pattern*

The core of the virtual reality application is the scene component. This provides:

- an information store; a consistent way of accessing any information in the application.
- registration; a way of keeping track of other components and the objects associated with these components.
- actions; each object registered with the scene can elect to run scripts when particular events occur, including an event which occurs on every frame for regular activity updates.

The hardware devices associated with a virtual reality system are also components of the application. There is typically an input system that receives stimuli from the user through controllers, keyboards, mice and other devices. Specialist input equipment such as cameras may also have their own particular components. Various forms of output components include the rendering system used to produce visual output, as well as audio and haptic output if used. These are accessed directly in some cases as standalone components, but may also have some specific functionality hooked into related objects in the scene such as controllers and cameras.

Objects in the virtual world fall into various categories:

- passive: these include objects which are static (unmovable) or elements that may be able to move but have no individual behaviour beyond that imposed through external control (such as a collision response generated by the physics system). These objects are completely defined by the information stored as part of the scene representation.
- active: these objects include custom scripts that extend their behaviour. While the information associated with these objects is maintained by the overall scene component, it is preferable to logically represent them as separate components in the virtual reality application pattern so that the interface (the services they provide to others, and require from other components) that they present to other components can be clearly distinguished and succinctly represented.

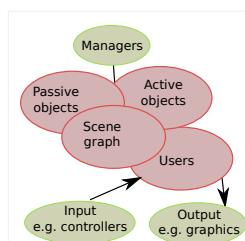


Figure 1.1.1: Typical structure of virtual reality applications.

- users: users are active objects which respond to input from the input system. They may be aggregated objects constructed from several other objects (controllers, cameras, avatar body objects) but it is helpful to consider how the user responds as a whole to events in the system rather than delegate this to each of the parts.

Some activities in a virtual world are not associated with any particular object but rather require consideration of information from multiple sources at once, or at a global level. These are often handled by managers. Managers may also register with the scene graph (to obtain some processing time) but often has an invisible representation. Managers are responsible for tasks such as collision detection, physics simulation, or network synchronization.

1.2 Manipulation

1.2.1 Description

One view of virtual reality is that it is a user interface (or user experience) paradigm. If that is the case then the key role of a virtual reality experience is to provide a way to allow users to manipulate computational constructs in order to achieve particular goals. For example, a virtual reality experience is just an interface to allow a game to be played more effectively, to allow data to be visualized and interacted with directly, or to manage some underlying software process or physical system mediated through the virtual reality overlay.

The ways of interacting with virtual worlds have not yet reached a stage of being completely standardized. There are some benefits in standardization; users can transfer skills to new applications and become immediately productive. However some of the benefits of the various forms of virtual reality and the range of hardware available to support them is that novel forms of manipulation are still being developed. A more valuable skill at present is to be able to conceive of, design, develop and evaluate forms of interaction with, and manipulation of, virtual worlds that make most effective use of the technology in the specific context of the virtual reality application that you are developing.

The patterns described represent some of the ways in which manipulation strategies of virtual worlds have been designed.¹

1.2.2 Pattern

Typical manipulation operations in a virtual world involve selecting items, or invoking actions on items that may have been selected. A range of patterns exist to perform such manipulation.

Manipulation pattern: Action at a distance. The controller is able to make things happen at distances beyond

¹ Some sources of guidelines for designing virtual reality interfaces:

- Oculus: <https://ocul.us/2R9YR7V>

the length of the user's arm. Typical examples of this is to cast a ray from the controller; a laser beam that is used to indicate which object to select, or which control is activated when a button is pressed. Other options include extensible arms. When the arm is slightly extended then it reaches nearby locations but as it extends further its effective reach is increased out of proportion of the length of the arm. This enables fine control over nearby objects, but also the ability to "touch" distant objects.

Manipulation pattern: Virtual reality menus. The menu systems used in typical 2D user interfaces are extended to 3D virtual reality versions. These may be shown as a collection of 3D buttons, or as a set of text options displayed one above the other on a convenient surface (including on the virtual representation of the controller itself). A touchpad or joystick can then be used to highlight a desired option and a trigger button pressed to activate it.

Manipulation pattern: Tools. The virtual representation of the controller is changed to resemble a specific tool e.g. hammer, camera. The actions of the controller in response to particular button presses is then dictated by the affordances of that particular tool. The controller continues to operate as that tool until reset, or a different tool is selected. Tool selection is performed via other selection techniques (ray-casting, or menus).

Manipulation pattern: Hands. The virtual reality controllers, particularly if using data gloves, are configured to look and behave as human hands. Naturalistic gestures can then be used to achieve particular forms of manipulation. Pointing at an object could, for example, correspond to selection. A closed hand is used to indicate an action such as picking up an object.

1.3 Locomotion

1.3.1 Description

Movement around a virtual world takes on different forms, often constrained by the space available in the physical setting and on the mobility of the participant encumbered with the virtual reality equipment being used. Locomotion is also associated with many of the hazards of using virtual reality applications, including motion sickness, and opportunities for physical injury.

As with manipulation of virtual environments, the patterns below represent some of the known design strategies for achieving effective, safe, and comfortable motion.²

1.3.2 Pattern

Humans can sense forces. Unbalanced forces manifest as acceleration and result in movement. A virtual reality application that supports locomotion needs to move a user, probably from some stationary location *A* to a stationary destination *B*. Any contiguous movement requires the user to get up to speed (accelerate) and then slow down again (decelerate). During this process the user does not feel any actual acceleration or force but only perceives it through the changing viewpoint in the virtual environment. This inconsistency can lead to motion sickness. It is even worse if the user does not expect it, so usually all motion should be initiated by deliberate action from the user.

Locomotion pattern: Physical movement only. The virtual reality experience does not require the user to move anywhere. The user is placed in a virtual world where all controls are within reach, or where manipulation (section 1.2) allows action at a distance. Where the equipment allows, the user may be able to get up and physically move around. Since both physical and virtual motion coincide, this should reduce likelihood of motion

² Some sources of guidelines for designing virtual reality locomotion:

- <https://ocul.us/2Ap3xBB>
- <https://ocul.us/2CAu5B6>

sickness due to this movement.

Locomotion pattern: Teleportation: The user designates a destination point, and triggers a teleportation operation. They are then instantly relocated in the virtual world to that destination. This is a popular option since the user initiates the movement, and there is no attempt to present any form of movement between the two points. The user often needs to reorient themselves at the destination so visible landmarks, ideally visible from source and destination, would help them establish where they are with respect to the world.

Locomotion pattern: Walking, Driving or Flying: These patterns allow the user to progressively move from one point to another. A walking approach allows the user to initiate movement and proceed at a constant pace in an indicated direction. The direction of movement is defined by pointing one of the controllers, or by moving in the direction of gaze. Acceleration effects (or lack thereof) occur when starting and stopping the movement, and when changing direction. Driving places the user in a vehicle so they have a fixed reference around them, and they can control direction and speed by operating the vehicles controls. Flying is more adventurous, removing constraints such as remaining fixed to the ground, or remaining upright. All of these patterns have potential for inducing motion sickness. There are suggestions that blanking out the peripheral image during this process can help reduce visual discomfort from rapid movement. The view can either be darkened, or all but the central part of the image replaced with black pixels. Alternatively a second version of the world that moves in the opposite way to the user is superimposed on top of the view of the moving world, to “cancel” out the feeling of movement.

Locomotion pattern: Tilt. This pattern involves the user tilting their head (or alternatively, their hands) with movement happening in the direction of the tilt. The degree of tilt could be related to the speed of movement,

allowing the user to control both direction and speed. Since this requires action on the part of the user, and because there is a relationship between movement of the head and the perceived motion, this technique could be expected to address some of the disparities between perceptions and physical stimuli. However it does constrain the user to maintain a vertical head posture at other times during the virtual reality experience.

Locomotion pattern: Arm running. Some applications have adopted a movement process whereby the arms are swung during movement. This mirrors the leg movement that would occur during a normal walking process in physical reality. The rate at which the arms are swung corresponds to the pace of movement. As with tilt, the expectation is that having some physical action involved helps reduce the effect of inconsistent visual and physical stimuli. A related approach to locomotion involves grabbing the world and pulling it past you. This helps justify to the user that they are remaining static in both physical and virtual reality, and that the motion forces are being applied to the world instead (with apologies to Newton's third law).

2

Representing and Manipulating Worlds

A virtual world is represented as information, as a data structure stored within the VR engine. Manipulation of the virtual world (such moving the user, picking things up, creating and altering content) is achieved by altering this state information. The users get to perceive the virtual world when the engine translates this data into a visual form through its rendering system. Rendering virtual worlds can also produce other modalities such as audio or haptic representations but, as with visual, output is largely controlled through existing processes in the VR engine.

The consequence of this is that developing a virtual reality application is equivalent to modifying these data elements at particular times or in response to certain events. The rest of this book explains:

- How the information content of a virtual world is represented as data.
- How to access and modify individual elements.
- How to identify what events occur and how to describe the responses to these.
- How to sequence a particular set of changes to the data representation to provide coherent actions and

interactions in a virtual world.

2.1 *Scene Graph: A Pattern for Representing Virtual Worlds*

2.1.1 *Description*

A virtual world consists of a collection of virtual objects. All these objects co-exist in a common space; the scene.

Objects also have relationships to one another. For example some objects may be attached to (or parts of) other objects. As such, when the parent object moves, the attached child objects also change location.

All objects have properties. All objects typically have information about their location (including position, orientation and size). Most objects have a visual representation of some kind to define their shape. Some objects may have additional components defining extra properties; such as how they respond to the physics of the virtual world, or specific behaviors unique to objects of that type.

The scene graph provides a way of representing all these details in a consistent and systematic fashion. As a side effect it provides a way of addressing every single piece of information stored in the graph i.e. it allows you to easily name each piece of information and to modify this information by referring to it by this name.

A graph consists of a number of nodes connected by edges. Graphically this is drawn as a number of circles (the nodes), connected by lines (the edges). A common type of graph is a tree where every node (apart from the root node) is connected by edges to one parent node, and to a number of child nodes. Most scene graphs have a tree-like structure although occasional edges may exist between two child nodes.

2.1.2 *Pattern*

A scene graph consists of a tree-like graph where:

- The root node represents the scene. The root node usually do not have a label.
- The child nodes of the root node are the objects in the scene. Each child node is given a label corresponding to the name of the object.
- Scene graph object nodes can have further child nodes representing objects that are attached to the parent node (e.g. wheel objects might be attached to the parent car-body object). These nodes are connected with a parent-child edge. These nodes are given a label corresponding to the name of the child object.
- Scene graph object nodes can have child nodes attached that represent particular functional components (e.g. a component that represents location of the object, or that representing the geometrical shape of the object). These nodes are connected with an object-component edge. These nodes are given a label corresponding to the name of the component.
- Components can have particular properties. A location component likely has a position property, an orientation (rotation) property and a size (scale) property. Component nodes are connected to property nodes with a component-property edge. These nodes are given a label corresponding to the name of the property.
- Some scene graphs may have sub-component (component has several child components) or sub-property (property is a collection of several child properties) nodes.

The properties hold the information that defines the virtual world. The scene graph itself describes the relationships between the individual data elements.

An individual property is named by concatenating all the labels in the path from the root node to the node corresponding to the property.

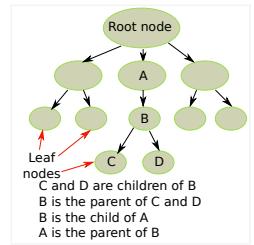


Figure 2.1.1: A graph (nodes and edges) arranged in a tree structure.

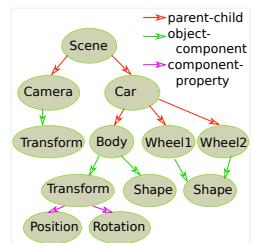


Figure 2.1.2: Some of the different categories of relationship between nodes in a scene graph.

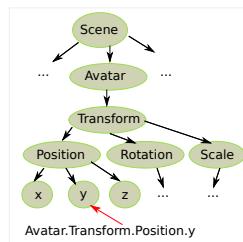


Figure 2.1.3: The names of properties are determined by the path in the scene graph leading from the root node to that particular property.

In practice different VR engines may have specific conventions for using a particular label. For example, retrieving a node A after following an object-component edge may require the name to be represented as $\text{GetComponent}(A)$.

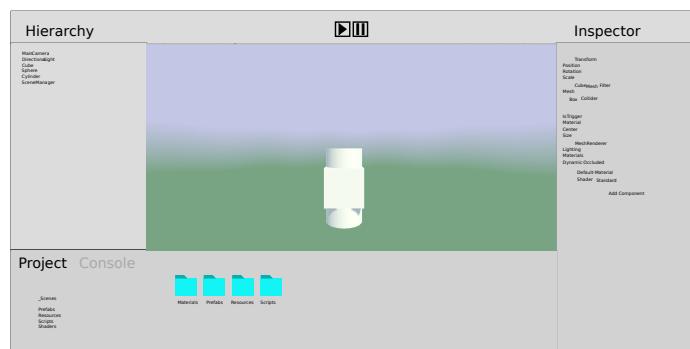
2.1.3 Example

The examples provided contain instructions that you can follow to recreate the results shown. These do expect some level of familiarity with the particular virtual reality engine being used, and access to the manual for the engine would be beneficial for those new to the engine. It is worth completing the introduction in section 11.1 on page 353 to first learn the steps corresponding to each of the processes described in these instructions.

Example 1. Applying the pattern to Unity software

1. This example demonstrates how we can assign particular values to properties of objects in the scene graph to modify the virtual world. Start with a new project and create the initial scene shown in Figure 2.1.4 which contains a cube, sphere and cylinder all with default property values in addition to the default scene elements. They all end up overlapping one another (the sphere is hidden inside the cube but is seen in the scene graph listing on the left).

Figure 2.1.4: The initial configuration, containing a cube, sphere and cylinder all with default values and at the same default position.



2. An empty node called SceneManager is added. A C# script component called PropertyModifier is added to this. The script contains instructions to be executed one after the other when the application is started, to make the following changes to properties of elements in the scene.

The script starts off by searching the scene graph for nodes with the names of the objects we're after. If this strategy is used in a virtual reality application then it is important that care is taken with naming of objects to avoid duplication or reuse of names.

Property modification instructions are shown in Algorithm 1 on the next page. The objects that are identified provide the details of the path from root of the scene graph to object level. Object to component links are followed with the GetComponent<ComponentName>() instruction¹. The properties of the component are then identified with the .property notation. Note the convention of ending a floating point constant values of type float with the letter f. This distinguishes the constant value from another similar type in C#, the double which has higher precision but requires more storage. Position vectors and other common types in Unity software favour the use of float.

3. Run the application to produce the results shown in Figure 2.1.5 on page 27. The original three elements remain in the scene, but the rendered view of the scene is significantly different despite only having manipulated an information-based representation of the world.

¹ object.transform is a shortcut to the transform component because it is used frequently. In practice GetComponent<Transform>() does the same thing.

2.2 Maintaining Objectivity

2.2.1 Description

The word “object” has multiple meanings when it comes to developing virtual reality systems. The different inter-

Algorithm 1 Properties of the scene are modified by finding the path through the node and component links until reaching the property. The information in properties can usually be examined and modified.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PropertyModifier : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9          GameObject cubeObject = GameObject.Find ("Cube");
10         GameObject sphereObject = GameObject.Find ("Sphere
11             ");
12         GameObject cylinderObject = GameObject.Find (""
13             Cylinder");
14
15         // Move the cube 2 units to the left.
16         cubeObject.transform.position = new Vector3 (-2.0f
17             , 0.0f, 0.0f);
18         // Move the sphere 3 units upwards.
19         sphereObject.transform.position = new Vector3 (0.0
20             f, 3.0f, 0.0f);
21         // Set the cylinder material colour to red.
22         cylinderObject.GetComponent<Renderer>().materials
23             [0].color = new Color (1.0f, 0.0f, 0.0f);
24         // Copy the shape of the cylinder to the shape of
25             the cube.
26         cubeObject.GetComponent<MeshFilter>().mesh =
27             cylinderObject.GetComponent<MeshFilter>().
28                 mesh;
29         // Rotate the cube object (which now looks like a
30             cylinder).
31         cubeObject.transform.rotation = Quaternion.
32             AngleAxis (45.0f, new Vector3 (1.0f, 1.0f,
33                 1.0f));
34     }
35
36     // Update is called once per frame
37     void Update () {
38
39     }
```

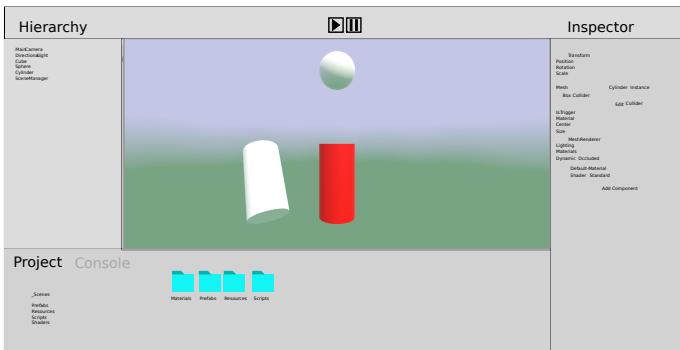


Figure 2.1.5: The appearance of the rendered scene after a number of properties have been changed.

pretations have some degree of overlap but also sufficient variation to ensure confusion when used in contexts that do not include the overlap.

An object is used in the sense of a visible element of the virtual environment. Here the key characteristics of an object is that it has a number of properties, particularly a geometric shape, associated with it. Properties common to all objects present in the scene graph include a transformation, representing as a position, rotation and scale value, potential links to parent and child objects, and a set of components which in turn contain particular properties. Objects can also be responsive in that they do things when triggered by events occurring in the virtual world, and autonomous, in that they undertake actions based on behaviour particular to that element (without having to be triggered).

An object can also exist as a programming concept under the paradigm of object oriented programming. This involves decomposing applications into a number of independent but interacting components called objects. Each object consists of attributes: data values specific to that instance of the object, similar to the concept of properties, and methods: functions that perform particular operations on that object, similar to the responsive and autonomous behaviours. Objects interact with one another by invoking methods on other objects. The subset of methods that are visible and available for other objects

to use is the public interface of the object.

The object-oriented paradigm simplifies the process of creating virtual reality (and other) applications by reducing the complexity into simpler elements, each with clearly delineated responsibilities (an object is only concerned about the functionality associated with that object), and a restricted interface which represent the only interactions that it has to support with other objects.

Object-orientated objects are developed by inheriting and extending the attributes and methods of a base class. This avoids having to repeat all the work involved in developing the base class but allows additional or modified functionality to be included in the derived class. Commonly used configurations have been identified as program design patterns [Gamma et al., 1995]. The virtual reality patterns used in this book are loosely based on this design process.

Virtual reality engines incorporate both interpretations of the object concept into the process of developing virtual reality applications. Geometric objects that are added to the scene graph are also object-oriented objects that inherit their common properties from a engine defined base class. The additional components added to extend the capabilities of the objects in the scene graph are additional objects, derived from a Component base class, in accordance to the Component design pattern [Nystrom, 2014]². Each component can define functions that are called from other components on other scene graph objects, or directly by the game engine.

² <http://gameprogrammingpatterns.com/component.html>

2.2.2 *Pattern*

The template for an object is known as a class and contains all of the definitions relating to attributes, code structure, details of the interface provided, and how the class inherits from other classes. The class is purely a description so to use this structure in an application it must be instantiated as an object. An new instance of a

class is created in most languages using the new operator. In a virtual reality application, the instantiated object must then also be added to the scene graph (or as a component of an object in the scene graph) as described in section 2.3 on page 33.

The objects added to the scene graph have a class definition pattern along the lines of:

```
class SceneObject
{
    Attributes:
        List of components
    Functions:
        function setup ()
        {
            addComponent (new transformation
                          component)
            addComponent (new geometric shape
                          component)
        }
        function update (deltaTime)
        {
            for each component
            {
                component.update (deltaTime)
            }
        }
        function addComponent (component)
        {
            components.add (component)
            component.setup ()
        }
}
```

This class provides the template for a new scene element added to the scene graph. While all of this functionality is internal to the virtual reality engine, this pattern represents conceptually what happens when a new object is instantiated. Specifically the object keeps track of a list of its components, has a setup function called by the engine to initialize the object which adds some standard components, and has an update function called by the engine for each frame which then calls an update function in each component to give it a chance to run. This is in case there is some autonomous behaviour which needs to take place on a regular basis.

We add functionality to scene objects by adding components to them. The components adhere to a particular component structure enforced by inheriting from a base component class. Typically this requires components to provide a public interface consisting of at least an initialization (setup) function, and a frame update event handler (update) function.

```
class OurComponent inherits BaseComponent
{
    public:
        // Also any public properties for
        // this component.
        function setup ()
        {
            ...
        }
        function update (deltaTime)
        {
            ...
        }
}
```

The custom component can also include a number of public properties. Many game engines make public properties available for inspection and manipulation directly

in the engine's editor. Components can also have private properties and functions that exist purely to support the internal operations of the component.

2.2.3 Example

Example 2. Applying the pattern to Unity software

1. This example adds a component to an object in the virtual world. That component provides public properties that are manipulated from the editor, even while the application is running. It also makes use of both the start, and update event handlers that each script in Unity software gets when the script is created. Start with a new or existing scene, and add an element to the scene (say, a cube).

Add a new C# script to the project, named ChangeObject. Drag this script onto the element in the scene we created for it. For the first step, we change the colour of this object in the start function. This function only runs once when the application first starts running.

Include the following line in the body of the start function.

```
1  this.GetComponent<MeshRenderer> ().material.color =
   new Color (1, 0, 0);
```

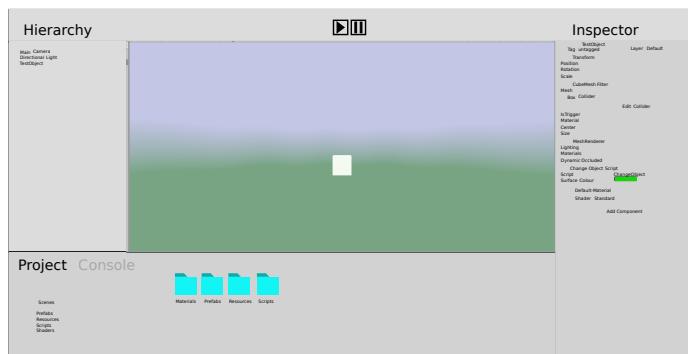
The final version of the script is shown in Algorithm 2 on page 34 for your reference. Run the application. The chosen object should turn red as soon as the application starts.

2. The next step is to provide a public property for this component. Public properties are added to the set of properties available for each component within the Inspector panel (and might start to suggest that all of the existing components of an object provided by the engine are created through similar looking scripts). Above the start function but still inside the class, add the line:

```
1 public Color surfaceColour = new Color (0, 1, 0);
```

This creates a variable named `surfaceColour` of type `Color` (a Unity software specific type) and makes it public. It also sets the initial value of the variable to green (the three parameters represent amount of red, green and blue respectively). As soon as you save the script (and assuming there are no errors) you should see an extra property added to the `ChangeObject` component in the Inspector as shown in Figure 2.2.1.

Figure 2.2.1: The public variable `surfaceColour` is now available through the Inspector pane.



3. If we run the program, the object still insists on turning red. This is because the code in the `start` function is unchanged, and in particular doesn't make any use of the `surfaceColour` variable we have defined. Change the line to read:

```
1 this.GetComponent<MeshRenderer> ().material.color =
surfaceColour;
```

The object should now become green (or whatever value is in `surfaceColour`) when the program runs.

4. You should notice that if you change the value of `surfaceColour` in the Inspector before the program runs then this changes the colour of the object when the program runs. If you change `surfaceColour` while the program is running it has no effect (and the change is also reset by Unity software when the program is

stopped). This is because the start function is only run for a single instant when the program starts.

Copy the line to set the material colour and add it into the update function as well.

Now when we modify the colour value in the Inspector pane the object changes immediately. This is because the update function is being repeatedly run on every frame - more than 30 times a second.

5. We can emphasize the repeated action of the update function by replacing its contents with this instruction:

```

1  this.GetComponent<MeshRenderer> ().material.color =
2    new Color
3      (Mathf.Abs (Mathf.Sin (0.37f * Time.time)),
4       Mathf.Abs (Mathf.Sin (0.71f * Time.time)),
5       Mathf.Abs (Mathf.Sin (0.52f * Time.time)));

```

This makes use of Unity's time property to produce slightly different color values each time it is called.

Running the program now should show a constantly changing hue. The surfaceColour property now has no effect because the colour set in the start function is being continually overwritten in the update function. For those interested, the principle behind the periodic colour generation is described in section 9.2 on page 299. For reference, the complete script is shown in Algorithm 2 on the next page.

2.3 Adding objects to the virtual world

2.3.1 Description

There are several steps involved in creating new objects. New objects are usually an instance of a particular object template that has been previously prepared. Internal storage managed by the programming environment must be allocated for the instance. The new object must be registered with the scene graph so that it becomes visible in rendered images, and so that events generated in the scene can trigger functions in the object. The way the

Algorithm 2 Use of a public variable, and the start and update functions. Note the resemblance of this code to the pattern. This version uses Time.time to get the absolute time, rather than Time.deltaTime to get the time elapsed since the last update.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class ChangeObject : MonoBehaviour {
6
7      public Color surfaceColour = new Color (0, 1, 0);
8
9      // Use this for initialization
10     void Start () {
11         this.GetComponent<MeshRenderer> ().material.color
12             = surfaceColour;
13     }
14
15     // Update is called once per frame
16     void Update () {
17         this.GetComponent<MeshRenderer> ().material.color
18             =
19             new Color
20                 (Mathf.Abs (Mathf.Sin (0.37f * Time.time)),
21                  Mathf.Abs (Mathf.Sin (0.71f * Time.time)),
22                  Mathf.Abs (Mathf.Sin (0.52f * Time.time)));
23     }
24 }
```

object relates to other objects in the scene also needs to be defined, particularly for objects that participate in a parent-child relationship with other objects. By default, objects are placed immediately below the root level of the scene graph. Initial properties of the new object should also be set, so that it is initialized to a known state. The position and orientation of the new object should be set, as these are likely to be specific to each object created.

Some extra steps are required if adding objects to a multi-user virtual world distributed over a number of networked computers. Section 8.2 on page 276 provides the pattern for this.

2.3.2 Pattern

Assume that a template exists for the new object, called EntityTemplate. We may also have a reference to another object parentEntity that is the parent object for our newly created entity. The process of adding a new object to the virtual world makes use of the pattern:

```
// Allocate storage for a new object.
myEntity = new EntityTemplate
// Register object with the scene graph.
Register (myEntity)
// Optional: make new object a child of
// ParentEntity. Set the parent property
// of the new object.
myEntity.parent = parentEntity
// Set properties specific to the newly
// created object. Replace ◇ with
// appropriate values.
myEntity.position = Vector (◇, ◇, ◇)
myEntity.orientation =
    Quaternion (◇, Vector (◇, ◇, ◇))
```

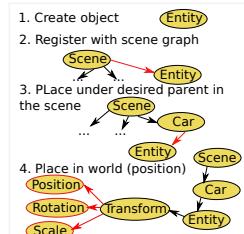


Figure 2.3.1: The steps involved in creating a new object and adding it to the virtual world.

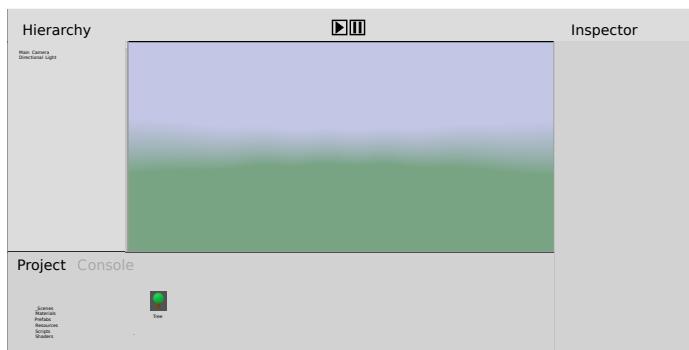
2.3.3 Example

Example 3. Applying the pattern to Unity software

This example demonstrates the process of using a script to create a new tree object instance.

1. [Figure 2.3.2] A template (prefab) for the tree is created by first setting up a number of primitives (sphere for leaves, and cylinder for trunk) in the scene, adding some cosmetic frills such as materials with appropriate colours, and dragging this to the project window. Prefabs are stored in a Prefabs folder to help keep the project organized as the complexity of the application grows. Remove the prefab from the scene after this, so that the scene appears empty, and the tree only exists in the Prefabs folder.

Figure 2.3.2: Initial configuration, with a prefab created to act as the template for a tree object.



2. We create a new Empty object to be responsible for the task of creating trees. Let us call this object the TreeManager. Add a new C# script to the TreeManager, called InstantiateObject. The code in Algorithm 3 on the facing page represents the Unity software and C# adaptation of the pattern provided above. Note that several steps of the pattern have been combined into the single Instantiate function provided by Unity software. Extra parameters to this call include the initial position (section 2.4 on page 38) and orientation (section 2.5 on page 43) of the object.

The Start function creates a variable called `objectInstance` that holds the reference to the newly created object. We haven't bothered to preserve this value

Algorithm 3 The object instantiation pattern for Unity software and C#.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class InstantiateObject : MonoBehaviour {
6
7      public GameObject objectTemplate;
8
9      // Use this for initialization
10     void Start () {
11         GameObject objectInstance = Instantiate (
12             objectTemplate, new Vector3 (5, 0, 2),
13             Quaternion.identity);
14         objectInstance.transform.parent = null;
15     }
16
17     // Update is called once per frame
18     void Update () {
19 }
```

when Start comes to an end, since we would be able to retrieve this value from the scene graph should we ever need it later. However if we intend to make frequent use of this object then it would be worth assigning this value to a longer lived variable.

3. In the Unity software inspector window, assign the tree prefab to the Object Template field of the TreeManager.

4. [Figure 2.3.3 on the next page] When run the tree appears off to the side given that its position was set to (5,0,2) - off to the right and slightly further back from the camera. This script has no specific dependency on our tree prefab, and could be used to instantiate any object, from any prefab provided to the objectTemplate variable.

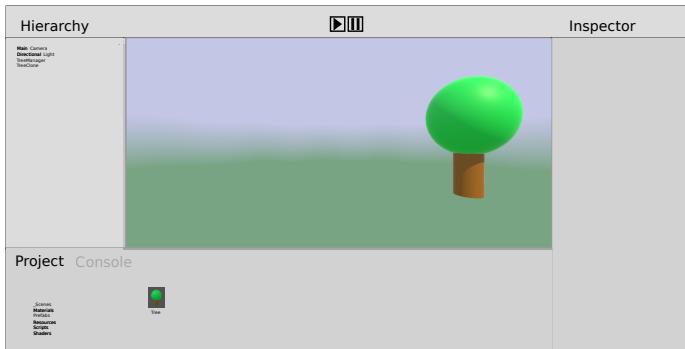


Figure 2.3.3: The tree shown at the position it was instantiated at.

2.4 Vectors: Representing Position and Scale

2.4.1 Description

Vectors get used interchangeably in two different ways during virtual reality application development:

- Virtual reality and games: a vector is a quantity representing direction and distance in space. Typically a Cartesian coordinate system is used, with x, y and z axes, and a vector representing a displacement in space has three components corresponding to the change in the x-direction, y-direction and z-direction respectively.
- Programming: vectors are a list of numbers.

During development of virtual reality applications, vectors are often a list of 3 numbers. In some cases, these 3 numbers may correspond to the x, y and z components of a displacement vector. In other cases they may be used to represent groupings of 3 other spatially related values, such as the relative scale of an object in each of the 3 axis directions.

2.4.2 Pattern

The following are some of the most common patterns in which a vector is used:

- Position: the three element vector represents the position in space defined by the x, y and z coordinates provided. For the purists: the vector represents a displacement from the origin of the coordinate system along each of the three coordinate basis vectors. The “relative to the coordinate system” aspect is quite important: parent-child edges in the scene graph generally imply that the child position is relative to the coordinate system of the parent. This implies that as the parent is transformed through space that the child object maintains its fixed position relative to the parent, and moves with it.
- Orientation: the three element vector represents rotations around the x, y and z axes respectively. This is a horrible representation of orientation but unfortunately one in common use on many platforms. It does also depend on the order in which the sequence of rotations are applied, and this may depend on choices made by the developers of the VR engine that you are using.
- Scale: the three element vector represents the amount of stretch (or shrinkage) of the object relative to each of the x, y and z axes. As a rule of thumb you should scale by the same amount in each direction (uniform scaling). If you do use non-uniform scaling then ideally apply this only to leaf-node (the last child object in the parent-child sequences) even if you have to create an extra child node to do so. Otherwise you may be surprised by the effects achieved when children inherit this distortion from their parent objects.
- Velocity, acceleration and force. These are vectors used for the kinematic and dynamic effects within physics simulation provided by the VR engine.

Variations on this pattern do also occur:

- Colour: the three element representation of colour uses the values to describe the red, green and blue

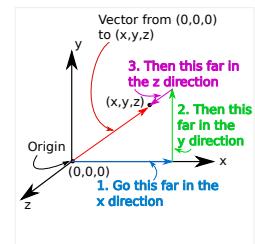


Figure 2.4.1: The x, y and z coordinates represent the distances along the x, y and z axes that a given location is from the origin of the coordinate system.

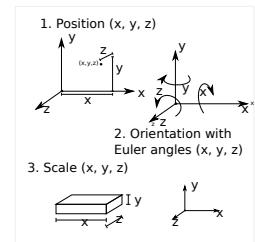


Figure 2.4.2: A vector of 3 elements is variously used to represent position, orientation and scale, as well as other quantities relevant to virtual reality systems.

intensities of the colour in most cases. Other three element colour representations do exist, such as hue, saturation and value. Four element representations might also occur, with the last element representing alpha; the degree to which the colour covers the object or pixel. The alpha value is usually used to achieve transparency effects.

2.4.3 Example

Example 4. Applying the pattern to Unity software

1. This example demonstrates the use of vectors to manipulate the position and scale of objects. Unity software has several vector types. The one we use is `Vector3`, a three component vector that has fields for x, y and z coordinates.

Start with a new project. Add a cube at the origin.

Create a C# script, name it `ApplyVectors`, and add it as a component to the cube object.

Now add a cylinder object. Set its position to 0,2,0 and in the scene graph (Hierarchy) view drag it over the cube object so that it ends up indented slightly below the cube. This makes it a child object of the cube. The cube is then the parent of the cylinder.

2. Edit the `ApplyVectors` script to match the code shown in Algorithm 4. The script identifies the position and scale fields of the object to which the component is attached. The chain is read as: this (the `ApplyVectors` component object) followed by `gameObject` (the object that the `ApplyVectors` component is attached to) followed by `transform` (the `transform` component of the cube) followed by `position` (the `position` property of the `transform` component). The size is controlled by a variable called `localScale` since there are different coordinate systems that scale can be defined in (see section 2.6 on page 47). The values assigned to `position` and `scale` are both `Vector3` objects. When these

Algorithm 4 Vectors are used to set the position and scale of the object to which this component is attached.

```

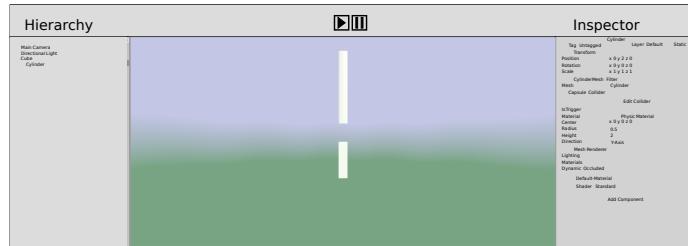
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class ApplyVectors : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9          this.gameObject.transform.position = new Vector3
10         (1, 2, 3);
11         this.gameObject.transform.localScale = new Vector3
12         (0.5f, 2.0f, 0.1f);
13     }
14
15     // Update is called once per frame
16     void Update () {
17 }
```

are constructed they are provided with 3 parameters corresponding to the x, y and z values respectively. Normally these must be floating point (float) types, and in C# constant values have an *f* suffix to identify them as such. Integers can be used but are implicitly converted to floats when the script is compiled.

3. When we run this we can expect to see that the cube moves to the specified position, and reshapes to the specified scale proportions. The cylinder undergoes some interesting changes of its own (Figure 2.4.3). As a child object it inherits the transformation of the parent, so also undergoes a change of position, maintaining its position relative to the new position of the parent. Its size changes as it also inherits the scale part of the transformation. The scale change is often unexpected or undesirable, particularly if additional transformations such as rotation are involved. Hence we tend to prefer scaling only objects that have no children. This might involve padding

the parent-child hierarchy with some invisible intermediate nodes (the role of the empty object in Unity software). This also indicates why Unity software would distinguish between a local scale (including the parent scale changes) or a world scale that ignores them.

Figure 2.4.3: The effects of applying a position and scale change to the parent object in a cube/cylinder hierarchy.



4. We try applying the same offset to the camera in the scene. Create a new C# script, name it ShiftByVector, and add it as a component to the Main Camera object. Include the line below in the start function:

```
1     this.gameObject.transform.position =
2     this.gameObject.transform.position + new
        Vector3(1, 2, 3);
```

This adds the same vector to the position of the camera, as was being applied to the cube to offset the position of the camera from where it started. For the pedantic, we didn't add the vector to the cube's position, but since it started at 0,0,0 the net effect is the same.

When you run the application the cube seems to remain unchanged. However the change in size confirms that the transformation is still applied. What has happened is that the camera has moved by the same amount. This demonstrates a particularly significant way of understanding cameras: they can either be objects in the scene transformed just like everything else (a world perspective), or they are special objects whose transformations seem to invert or cancel out the same

transformation applied to other objects when viewing the world through the camera (the camera/view perspective).

Moving the camera around could be achieved by either changing the position of the camera, or by changing the position of everything else in the world in the opposite direction. The latter alternative may seem inefficient but is one valid interpretation of what actually happens in the transformation pipeline used in virtual reality systems (see section 2.6 on page 47).

2.5 Quaternions: Representing Orientation

2.5.1 Description

There are several ways of representing the orientation of objects in a virtual environment.

A common approach is based on Euler angles, the amount of rotation about the x, y and z angles respectively. The approach is dependent on the order in which the 3 rotations are applied. Since there are 6 valid permutations achieving consistent results across different platforms using this approach is difficult. Euler angle representations also suffer from gimbal lock, where incremental rotations may end up in scenarios where axes line up and results in a scenario where it is non-trivial to continue to rotate in particular directions. As a result, Euler angle representations for manipulating orientation are avoided.

In all cases any angles and directions are relative to the rest position of the object, and the origin and axis directions in the current coordinate frame. It is worth keeping this point in mind when working with parent-child hierarchies as this coordinate frame may be subject to the movements of the parent object.

An alternative approach is to represent changes of orientation as a single rotation by a given angle about a particular direction in space. This is associated with a

rather elegant mathematical representation of rotations, called quaternions. The specific representation of a quaternion has its own internal representation but, for the purposes of using a VR engine, can be considered equivalent to defining a unit vector in space (the direction of the rotation axis), and the angle representing the amount of rotation. Quaternion representations are used to define incremental rotations of an object, or an absolute orientation specified as the rotation from rest position to current orientation.

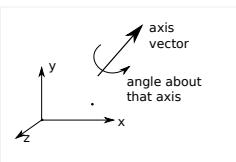


Figure 2.5.1: The angle-axis representation of orientation.

Incremental transformations are normally applied by multiplying the current transformation by them to produce the new absolute transformation to be applied to the object. Thus the new orientation is the product of the old orientation and the incremental rotation specified as a quaternion.

It is worth noting that, depending on the VR engine used and the context within that engine, that angles may be either represented using degrees ($^{\circ}$) or radians (c). The key difference between them is the range: degrees cover the range $[0^{\circ}, 360^{\circ}]$ while radians have a smaller range of $[0^c, 2\pi^c]$. Degrees are often used in GUI style property managers, while radians are typically used during scripting with mathematical libraries (such as trigonometric functions: sin, cos). The convention used is usually specified in the documentation for the property or function.

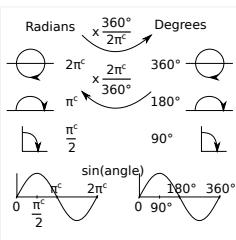


Figure 2.5.2: Radians and degrees represent the same concept of angles, just on a different scale.

An alternative orientation representation is to define three vectors representing the up, forward and right (or left) directions for the object relative to the world's coordinate frame. This can simplify common operations as these provide the direction vector required for quaternions representing common rotations. For example, making an object turn left or right is achieved with positive or negative angular rotations about the object's up direction. The observant will note that these three vectors are the basis for the object's local coordinate system expressed in world coordinates.

Many VR engines maintain a single internal representation of an object's orientation while allowing this to be presented and manipulated using any of the alternative equivalent conventions discussed in this section.

2.5.2 Pattern

Rotation:

```
object.transform.orientation =
    object.transform.orientation *
        Quaternion (angle, Vector (x,y,z))
```

Converting between degrees and radians:

$$\begin{aligned} \text{radians} &= \frac{2\pi}{360} \text{degrees} \\ &= \frac{\pi}{180} \text{degrees} \end{aligned}$$

$$\begin{aligned} \text{degrees} &= \frac{360}{2\pi} \text{radians} \\ &= \frac{180}{\pi} \text{radians} \end{aligned}$$

2.5.3 Example

Example 5. Applying the pattern to Unity software

- [Figure 2.5.3] Assume we have a Unity software scene and we want to modify the orientation of a particular object using a script.

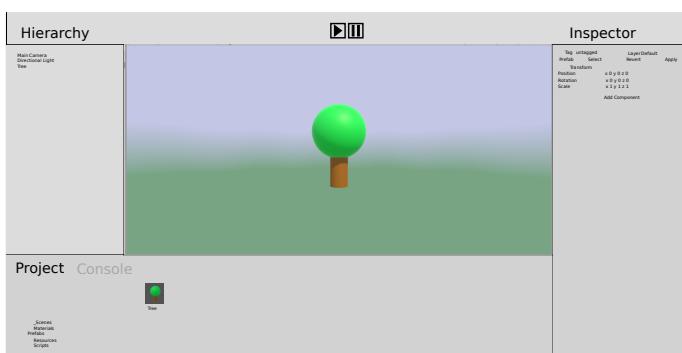


Figure 2.5.3: The starting configuration of our scene, with a tree-like object with default position and orientation.

Algorithm 5 Defining a new orientation to the current object by giving a rotation about a specified axis (the vector) by the angle in degrees.

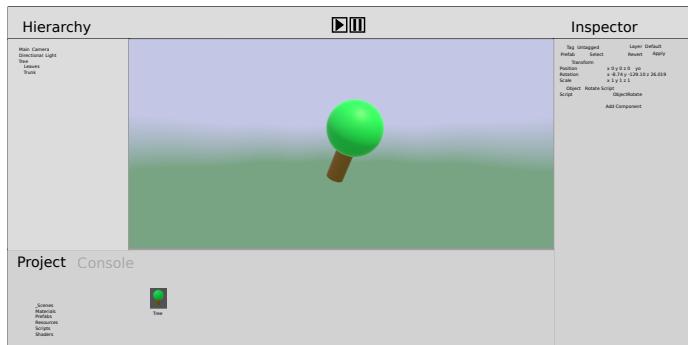
```

1     void Start () {
2         this.transform.rotation = Quaternion.
3             AngleAxis (37.0f, new Vector3 (0.4f,
4                 0.5f, -0.3f));
5     }

```

2. Add a C# script called ObjectRotate to the object we want to manipulate. Include the code shown in Algorithm 5. This code is in the Start function and only runs once when the program starts, to create the rotated object shown in Figure 2.5.4.

Figure 2.5.4: Change to rendered view, and scene graph properties after a new angle-axis rotation is applied to the object.



3. A regular rotation while the application is running is a convenient way of getting some autonomous action happening in a scene, to draw the user's attention to particular elements. The same process is used, except that the code is included in the Update function which runs every time a frame is redrawn. The quaternion is also multiplied by the existing orientation to accumulate the rotations, rather than just overwrite the previous orientation setting. The script in Algorithm 6 on the next page uses a smaller angle increment since this represents the angular change from one frame to the next (angular change in about $\frac{1}{30}$ s). Of the two

Algorithm 6 Continuous rotation through combining the current orientation with an incremental rotation.

```

1   void Update () {
2       this.transform.rotation = this.transform.
3           rotation * Quaternion.AngleAxis (0.7f,
4               new Vector3 (0.1f, 0.5f, -0.1f));
5   //      this.transform.RotateAround (new Vector3
6   //          (0.0f, 0.0f, 0.0f), new Vector3 (0.1f, 0.5f, -0.1
7   // f), 0.7f);
8   //      this.transform.Rotate (0.1f, 0.2f, 0.3f);
9 }
```

commented-out alternatives, the Unity software function RotateAround performs a similar operation using the angle and axis parameters, with an additional parameters specifying the point to rotate around (we use the origin since the object is currently centered on that). The Unity software function Rotate is less desirable since the parameters are the three Euler angles, representing angular change with respect to the 3 coordinate axes, but convenient for a quick and dirty test application.

2.6 *Where is everybody? Transformations, cameras and coordinate systems*

2.6.1 *Description*

The position and orientation values assigned to individual objects in the scene graph all get used during the rendering process to produce a graphical representation of the scene as seen from the perspective of the camera in the scene, and suitably projected so that the resulting image conforms to the perspective and resolution requirements of the display device. The process of working out where each element is positioned relative to all other objects is known as the transformation pipeline.

To understand the transformation pipeline it is necessary to be aware that the process of moving a rigid

virtual object to a given position involves a transformation known as translation (shifts the object from where it is to where it needs to go). Achieving a given orientation involves a process known as rotation. Rotation occurs about the origin of the coordinate system used, which means that to rotate an object about its centre we may need to translate the centre of the object to the origin (if it isn't there already), rotate, then translate back. Changing the size of the object involves applying a transformation known as scaling. Scaling also occurs relative to the origin of the coordinate system.

The transformation pipeline thus consists of a combination of many transformations and can be difficult to conceptualize in its entirety. To simplify this we have settled on a standardize decomposition of the pipeline into a number of predefined phases. Each phase involves setting up the scene in a particular coordinate system.

A coordinate system defines the space we are working in - literally the structure of a completely new universe. This should not be surprising given that the virtual realities we create exist in their own private pocket universes. The coordinate system defines a single unique point for our universe; its centre known as the origin. We also need to know a forward, sideways and up reference direction so we define three vectors that are all perpendicular to each other for this purpose. The coordinate system then allows us to meaningfully describe positions, orientations and sizes in terms of this information. For example, the location $(0, 2, 0)$ is achieved by starting at the origin and moving two steps in the direction of the second vector.

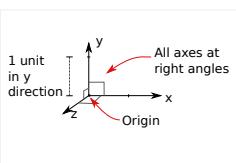


Figure 2.6.1: The coordinate system requires an origin, and three orthogonal axis vectors whose lengths define the scale of the pocket universe.

The scale of a coordinate system is somewhat arbitrary. There are no units naturally defined. It does help to agree on a particular convention that one then uses consistently through all parts of the application development process. A recommended convention is: 1 unit in the virtual world corresponds to 1 metre. Size and position of objects can then be set according to this.

A number of coordinate systems have been named:

Object coordinates: This is the coordinate system used by the 3D modelling package when it creates the geometry of an object that you're going to be working with. An intelligent modeller uses an object coordinate system with the origin in the centre of the object (or at ground level), and typically align symmetries of the object with the axis directions. For example, a human would usually be modelled with the origin at ground level midway between the feet, with the character facing in the position x-axis direction, the y-axis representing the upwards direction and left-right being in the z direction (Note: choice of which specific axis to use may vary). Morons exist and will place the model anywhere in space at arbitrary orientations which will typically need to be corrected through additional rotations and translations before any further transformations can occur.

In object coordinates, the individual object effectively exists in a universe of its own.

World coordinates: The scene in the virtual reality engine's editor typically shows the world coordinate frame. This is a common frame constructed to define the relative position, orientation, and size of each object in a common universe. The origin is chosen to be a point of significance to the world as a whole, such as a corner of a room if the virtual environment consists of a single room, or the centre of a city for a city oriented virtual reality. Axis directions are aligned with key directions in the world, such as the walls in the single room virtual world.

Placing objects in the world is regarded as initially overlapping the object and world coordinate systems and then scaling the axes of the object coordinate system so that the object is at the correct size, rotating the axes of the object coordinate system until the object is at the desired orientation, and translating the origin of

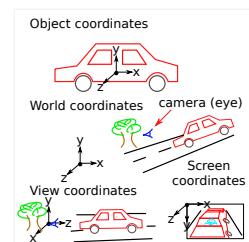


Figure 2.6.2: The transformation pipeline involves combining individual objects into a common world coordinate system, viewing this from a point of view of the camera and converting to a 2D image coordinate system.

the object coordinate system until the object is at the correct position. The translation offsets, the rotation angles, and the amount of scaling then become the position, rotation and scale properties of the object in the virtual reality engine.

Camera/View coordinates: One of the objects in the scene graph is designated as the camera. This is the object that defines how we perceive the world. It defines the position and orientation of a virtual camera viewing the scene, and corresponds to the viewer or head mounted display location in the virtual world.

The camera coordinate frame has very particular properties designed to simplify some of the stages of image production. The origin of this coordinate system is defined as the center of the camera (specifically the optical center or focal points for the perspective transformation). The coordinate axes use a convention such as: x-axis to the right, y-axis up and z-axis forward. The z coordinate of points in this coordinate system then indicates the depth of the point, or distance in front of the camera, which is exploited in other steps in the rendering process.

The transformation required to put the world into camera coordinates is usually the inverse of the one that put the camera in the world in the first place (i.e. from the camera's object coordinates to the camera's world coordinates). The key difference is that the entire world is affected by the transformation to camera coordinates (imagine picking up the entire universe and moving it so that the camera is now at the origin) and all objects in the world remain in the correct location relative to one another.

Screen coordinates: The view from the camera is then turned into a 2D coordinate system representing the image taken by the camera. This typically involves a projection transformation which flattens from 3D to 2D usually taking perspective into account. The origin

of this coordinate system may be one of the corners of the image, or its center. The x- and y-axes are aligned with the borders of the image. Some scaling may take place here to fit the image to the resolution of the device.

2.6.2 Pattern

The transformation pipeline is a standardized process within the rendering subsystem of the virtual reality engine. Control over parameters of the process is achieved by modifying the properties of the transformations:

Object to world: The position, rotation and scale properties of individual objects in the scene graph determine how the object coordinate system, in which the geometry of the object is defined, is transformed into the world coordinate system common to the scene.

World to view: The position, rotation and scale properties of the camera determine how the world coordinate system is transformed into the view coordinate system.

View to screen: The camera may also have properties that relate to perspective effects that are applied when the view coordinate system is transformed into the screen coordinate system. Properties of this image, such as the size or resolution can also be configured, or may be properties determined by external hardware devices.

2.6.3 Example

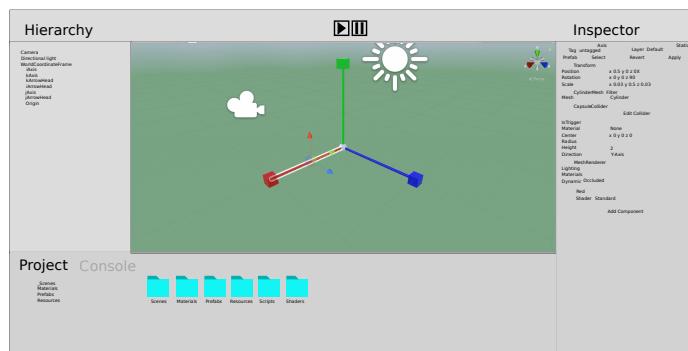
Example 6. Applying the pattern to Unity software

1. This example explores the different coordinate systems. To start off, we create a coordinate system object. This consists of a sphere to represent the origin, and 3 cylinders, capped with cubes (best approximation for arrow heads) to represent the 3 coordinate axes. We

make all of these components children of an empty object so that we can manipulate (transform) an entire coordinate frame just by considering this single parent object.

Start a new scene and create the structure described, and as shown in Figure 2.6.3. You may need to manipulate the scale of sphere and other elements. We want the empty object and sphere at 0,0,0 and each of the cylinders to be narrow but exactly one unit long. Note the Unity software cylinders are 2 units long by default and centred in the middle, so needs to be scaled by 0.5, and offset by 0.5. You can check if you've got things right by placing other objects at positions 1,0,0; 0,1,0 and 0,0,1 and make sure your axes reach from the origin to these points. We need several coordinate sys-

Figure 2.6.3: Defining an object hierarchy to represent a coordinate frame. The 3 axes are typically named \hat{i} , \hat{j} and \hat{k} corresponding to the Cartesian conventions of x , y and z respectively. We avoid using x , y and z since we are dealing with multiple frames which may not be aligned with the Cartesian world coordinates.



tems, so drag the compound object into your Project's Prefabs folder to make it into a template object. Then delete it from the scene.

2. Switch from the Game view to the Scene view so we don't get our example tainted by the presence of the camera. Delete the camera and light so the scene (universe) is completely empty. Drag one of the CoordinateFrame prefabs into the scene. Name this: ObjectACoordinateFrame. This represents the *object coordinate system* for an object we intend to eventually include in our virtual world. We'll add some vertices

for our object into this coordinate frame by importing and adding distinctive models. Make sure this object is a child of the coordinate system object so it shares the transformations of the coordinate system. Figure 2.6.4 shows the view of this particular universe.

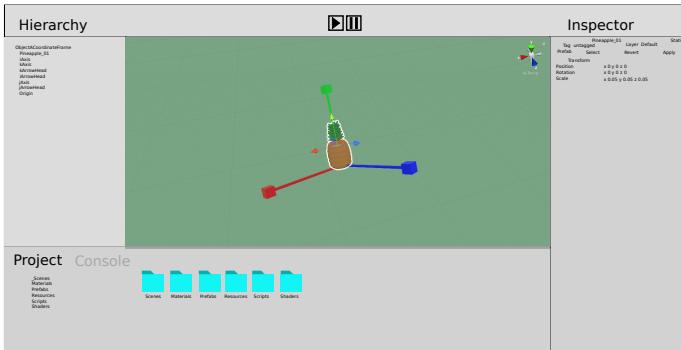


Figure 2.6.4: The object coordinate system for the pineapple object. Vertices in the pineapple are defined relative to the coordinate system shown.

3. We now want to populate a world. Add in another CoordinateFrame prefab, and name this one WorldCoordinateFrame. Without any world to object transformation, the world and object coordinate frames completely overlap. To make the world more interesting, add in a few more object coordinate frames (labelled B, C, D, ...) and attach individual child objects to each of them. Figure 2.6.5 shows how all of these objects all end up occupying the same space.

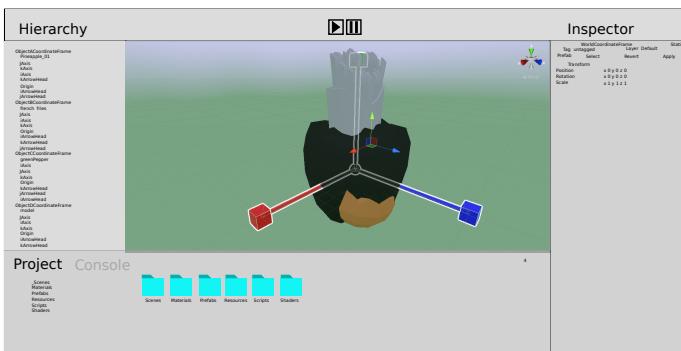
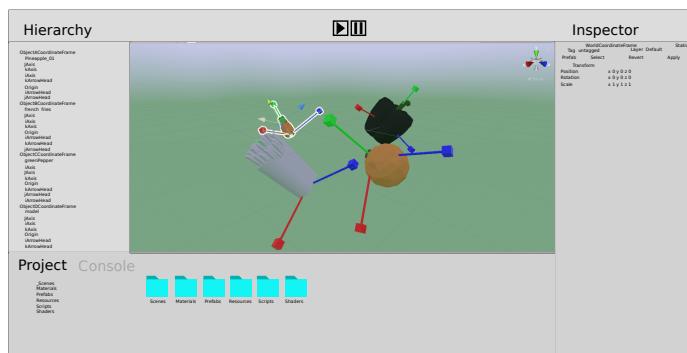


Figure 2.6.5: Multiple objects with their own coordinate system overlapping with the world coordinate frame since there are no object to world transformations.

4. We need to provide individual *object to world* transformations to place each object at an appropriate location in the world. Do this by selecting each object's coordinate frame (the empty object) and providing different values for the transform (position, rotation and scale) in the Inspector pane. Do not move the object representing the WorldCoordinateFrame; each object is being transformed relative to this and the object is just a visible marker for this. Figure 2.6.6 shows a potential world layout.

Figure 2.6.6: Each of the objects transformed relative to the world coordinate frame.



5. If we make each object coordinate frame object a child of the WorldCoordinateFrame object then we can see the effect of transforming the world as a whole. Modifying the transform properties of the WorldCoordinateFrame object moves the entire world. This is what is done when we perform the *world to view* transformation.

Add a new instance of the CoordinateFrame prefab to the scene as a child of the WorldCoordinateFrame and name this CameraCoordinateFrame. Add a camera object (using the Hierarchy Create button) and make this a child of the CameraCoordinateFrame. Make sure the transform properties for each of these elements are cleared, so that the camera coordinate system (the camera object's object coordinate system) is at the origin. Now transform the CameraCoordinateFrame

object relative to the world (the camera object's object to world transformation) to a suitable position and orientation so that it would be able to see some of the other object's in the world. Figure 2.6.7 shows the world consisting of multiple objects including the camera each with their coordinate frame transformed relative to the world coordinate frame (also visible).

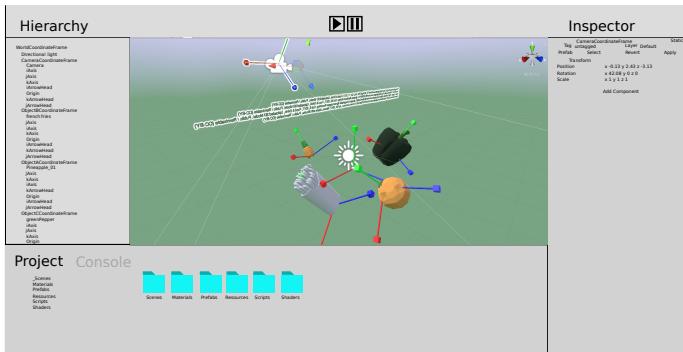


Figure 2.6.7: All objects in the scene, camera included, transformed into a common world coordinate system. The object representing the world coordinate frame is visible, as are the transformed frames for each of the objects inhabiting this world.

6. The view coordinate frame is the coordinate frame of the camera. The entire world (i.e. the world frame) needs to be transformed into this frame. We can view the effects of this transformation by selecting the Camera object. This provides a preview of the view of the world from the point of view of the camera's coordinate frame. You can also get the same effect by switching back to the Game tab, which shows the world from the perspective of the camera (Figure 2.6.8).
7. The rendered view shows the point of view of the camera, but also includes view to screen transformation. These transformations includes the perspective projection transformation required to produce the 2D image, as well as transformations to fit the rendered image into the window on the screen. Experiment with some of the camera settings that control the *view to screen* transformations.
 - The projection property of the camera; switch between a perspective projection and an ortho-

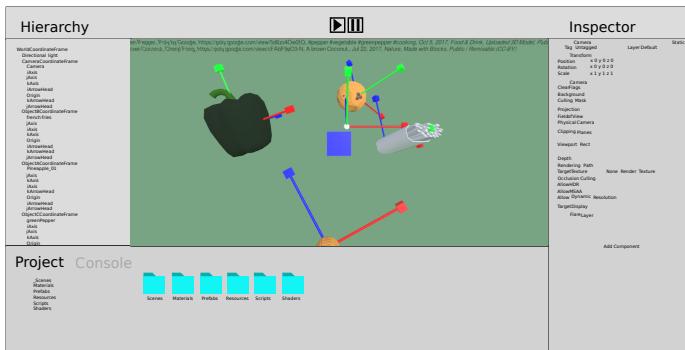


Figure 2.6.8: View of the scene in camera/view coordinates.

graphic projection (which does not adapt the size of the object with distance to the viewer).

- The field of view which dictates how much of the scene fits into the rendered window.
- The viewport rectangle which transforms the view of the screen to a portion of the visible window (example parameters: X=0.5, Y=0.25, W=1, H=0.5).

Example 7. Applying the pattern to Unity software

1. This example shows how to create a simple user controlled object for use when prototyping virtual reality applications. It is useful to be able to have an object that is driven around the scene. We can attach the camera to this object to create a user controllable viewpoint that is simpler to use when testing and developing VR applications than having to utilize the full head mounted display option.

This example makes use of the input manager pattern described in section 5.3 on page 179. We use this in preference to directly checking for key presses as it allows us to extend our control to include virtual reality controllers without making significant changes to the scripts we are working with.

2. Create a new scene, and add an object to represent the user. Make the camera a child of this object, to get the first person view from the perspective of this object.

You may want to position the camera at an appropriate position relative to the user object to either include the user's object in the view, or not as is appropriate to whether you want a third person or first person viewpoint. Being able to see the user's object is useful when you need to debug interactions between user and other objects in the scene.

You may wish to add other objects to provide some visual references in the virtual world. A floor plane is also useful since this example just navigates in the two horizontal directions.

3. Create a C# script called DriveableObject and add this as a component of the user's object. We assume you already have the input manager object from section 5.3 on page 179. You'll need to set the inputSystem variable of the DriveableObject in the Unity software Inspector window to this input manager.

Include the script from Algorithm 7 on the following page as the body of DriveableObject.

Forwards and backwards movement is achieved by adding copies of the object's forward vector to the current position. This vector is modified by:

- (a) The speed of movement; a public variable that is customised in the Inspector pane.
- (b) The amount of movement on the controller (variable v). This value becomes negative when moving backwards, so then subtracts the forward vector.
- (c) The time since the last update which ensures consistent speed regardless of capabilities of individual computers running your application.

Rotation is achieved similarly but in this case involves rotating by a angle about the up direction relative to the object.

Both the forward and up direction vectors are part of the representation of the object's orientation (section 2.5 on page 43).

Algorithm 7 A user controllable object that can update its position in the forward/backwards direction, and turn to the left or right.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class DriveableObject : MonoBehaviour {
6
7      public InputSystem inputSystem;
8
9      public float speed = 1.0f;
10     public float turnspeed = 0.1f;
11
12     // Use this for initialization
13     void Start () {
14         inputSystem.registerForInput (InputSystem.
15             ControlTypes.AxisAction, InputSystem.
16             ActionHandlers.MoveForward,
17             forwardEventHandler);
18         inputSystem.registerForInput (InputSystem.
19             ControlTypes.AxisAction, InputSystem.
20             ActionHandlers.TurnSideway,
21             turnEventHandler);
22     }
23
24     void forwardEventHandler (float v)
25     {
26         this.transform.position = this.transform.position
27             + speed * v * this.transform.forward * Time.
28             deltaTime;
29     }
30
31     void turnEventHandler (float v)
32     {
33         this.transform.rotation = this.transform.rotation
34             * Quaternion.AngleAxis (turnspeed * v * Time.
35             deltaTime, this.transform.up);
36     }
37 }
```

2.7 The Game Loop: Frame by Frame

2.7.1 Description

Patterns for designing and developing game loops can be found in other sources [Nystrom, 2014]. We assume the game loop is implemented in the VR engine and that our applications just need to make use of it.

2.7.2 Pattern

The typical game loop, from our point of view, has the structure:

```
initialize all objects
while the VR application is running
{
    handle input |
    update all objects |
    render scene graph
}
```

Depending on the complexity of the game loop implementation used in the VR engine we cannot assume:

- That objects are initialized in a particular order.
- That objects are updated in a particular order.
- That an update of objects occurs at the same rate as input is processed or individual frames of output are rendered.
- That the rate at which updates occur is constant, or even consistent.

As a consequence, our typical objects include the following pattern defining essential functionality:

```
VR Object
{
```

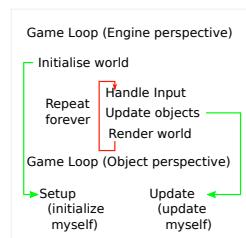


Figure 2.7.1: The game loop is standard across virtual reality engines although the application developer may only have to contribute particular portions when they include their own objects in the scene.

```
function initialize
{
    do all initializing of properties of
    this object.
    don't assume any other objects exist
    in the scene.
    don't assume any other objects are
    initialized.

}
function update (timeSinceLastUpdate)
{
    modify only the properties of this
    object to reflect changes that
    have occurred in the
    timeSinceLastUpdate.
}
}
```

There are patterns that are applied in cases where objects need to be initialized before others. However the VR application designer is advised to explicitly note this dependency.

2.7.3 *Example*

Example 8. Applying the pattern to Unity software

1. This example demonstrates the game loop pattern by building a timer or clock object for a virtual reality scene. This might serve as the basis for a countdown timer for use in a sports simulation, or other application that wants to let the user see how much time has elapsed. This timer may not be perfectly matched to real time since accumulating small time updates leads to loss of precision. Perfect time keeping is done by tracking the real-time clock in the computer. Here we instead use the time since last update value provided by the pattern.

Start with a new scene, and add a 3D text object which we use to show the timer. Call this object TimerDisplay, and create and add a new C# script component called TimerTracker. Add any additional geometry if you want to create a housing for your timer.

2. The game loop pattern is used as follows:
 - (a) When the initialize step occurs, we set the timer to its starting value.
 - (b) Each time the update steps occurs, we decrease the value in the timer. The amount to decrease by is based on the time that has elapsed since the last time the update event occurred.

The code in Algorithm 8 should be included in the TimerTracker script.

3. The timer includes a public variable which is made available as a property in the Unity software Inspector pane for that component. This makes the component slightly more reusable as it allows the initial value provided to the timer during initialization to be configured individually for each instance of the component.

The update step decreases the internal time variable by the elapsed time since last update provided in the variable Time.deltaTime. No check is currently made to see if the time reaches 0, and the result displayed in this case is not well behaved.

A separate showTime function is included to format the output and present it by modifying the text property of the 3D text object. This function splits the time counter into hours, minutes and seconds by converting it to an integer and using the remainder and divisor after dividing by the number of seconds in an hour, and number of seconds in a minute. The output produced is shown in Figure 2.7.2.

Algorithm 8 The game loop pattern applied to building a timer component.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TimerTracker : MonoBehaviour {
6
7      // The starting value for the timer. This will be
8      // set in the Unity software editor.
9      public float startingTimeValue = 120.0f;
10
11     // The current time value in seconds that the timer
12     // displays.
13     private float timeLeft;
14
15     // Use this for initialization
16     void Start () {
17         // initialize the timer value from the starting
18         // value provided.
19         timeLeft = startingTimeValue;
20         showTime ();
21     }
22
23     // Update is called once per frame
24     void Update () {
25         // decrease timer value according to the time that
26         // has passed
27         // since the last Update.
28         timeLeft = timeLeft - Time.deltaTime;
29         showTime ();
30     }
31
32     // Convert the timeLeft into hours, minutes and
33     // seconds
34     // and update the text display to show this.
35     private void showTime ()
36     {
37         int hours = ((int) timeLeft) / 3600;
38         int minutes = (((int) timeLeft) % 3600) / 60;
39         int seconds = ((int) timeLeft) % 60;
40         this.GetComponent <TextMesh>().text = hours.
41             ToString ("D2") + ":" + minutes.ToString ("D2"
42             " ") + ":" + seconds.ToString ("D2");
43     }
44 }
```

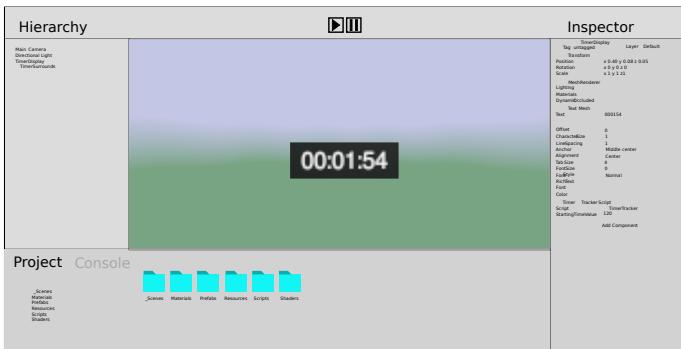


Figure 2.7.2: A view of the countdown timer running, also showing the scene graph structure and TimerTracker component with the public Starting Time Value property in the Inspector pane.

2.8 Having children

2.8.1 Description

The scene graph represents the variety of relationships between the information in the virtual world. For any object in the virtual world, just being attached to the scene graph in any way ensures that the object is visible in the rendered view of the scene (assuming that it should be visible and is not behind you, or behind another object). Objects have components and components have properties. However most scene graphs are visualized by showing the parent-child relationship that exists between objects in the scene graph.

This parent-child relationship is specifically intended to create a transformation hierarchy in the scene. Any child objects behave as if they were effectively anchored to their parent object. They adopt the position, orientation and scale of the parent object before applying their own position, orientation and scale *relative* to that. Behind the scenes, the transformation for the child object is combined with the transformation for the parent object to determine the child's position, orientation and scale.

Hierarchies are used frequently in a number of common scenarios:

- Any compound object (made up of lots of independent objects) is treated as a single entity by making

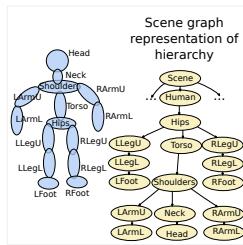


Figure 2.8.1: An articulated armature would use the parent-child relationships to construct the equivalent representation in the scene graph.

all parts children of a common parent. The complete object can then be manipulated directly by just operating on the parent object. The parent object need not even have a visible shape of its own and can just be a placeholder object used to organize things.

- Articulated body structures are represented as complex hierarchies, allowing each part to move relative to the piece that it is attached to. A typical human armature would use the hips as the root node, and attach torso and upper legs to this. Upper legs would connect to lower legs, which connect to feet, which connect to toes in turn. Similar structures would be used for the chains of bones corresponding to arms and head.
- A virtual reality application could have a user structure that includes a body armature as discussed previously. Various active elements would also be included. A camera (or two) may be attached to the head, and controllers may be attached to each of the hands.

A transformation hierarchy is normally driven top down. The root node (ultimate parent) is moved first, and its children are then positioned relative to that. This process continues with grandchildren and so on. This is a forward kinematics process. Since in a virtual reality system, the head and hands are often driven by information from motion trackers in the physical world, the process may need to run from children up to parents. This is an inverse kinematics process as described in section 7.3 on page 259.

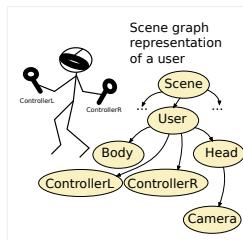


Figure 2.8.2: The user in a virtual reality application can use relationships in the scene graph to group all the nodes relevant to that particular user.

2.8.2 Pattern

The most common operations in a parent-child hierarchy include: making and breaking relationships, and identifying your parent and children. Each node may have only one parent, but can have an arbitrary number of children.

Identifying a parent object is done directly using the parent property. Some engines use the transform component of an object for creating parent/child links, rather than the object itself. So the parent object may be retrieved either as:

```
// if the parent property stores objects
childObject.parent
// or if the parent property stores
// transformations
childObject.parent.object
```

As in section 2.3 on page 33, adding a parent object just involves setting the parent property of the object.

```
childObject.parent = parentObject
// or if parent property stores transformations
childObject.parent = parentObject.transform
```

This automatically includes the childObject in the list of children of the parent. Thus adding a child object is done by just setting the parent property of the intended child.

Removing this relationship involves clearing the parent property.

```
childObject.parent = null
```

The name of a undefined reference may vary between programming languages. The result of removing a parent leaves the child object at the root level of the scene graph together with all the other objects without parents.

The list of children is retrieved using an engine specific function along the lines of getChildren. The list returned may be empty, contain just one element, or multiple elements.

```
List allChildren = object.getChildren ()
```

The list is handy if we need to perform a particular action on all children. Later sections describe patterns for working with lists (section 3.6 on page 115). However

sometimes we need to access a particular child. We need some way of identifying the child - a name or unique identifier. A function such as `findChildObject` can then be used to search through all children to return a reference to the desired one. If such a child does not exist then the null reference is returned.

```
childObject = object.findChildObject
    (childObjectName)
```

Depending on the engine used, this function may only search immediate children. Similar functions may exist for searching all children (i.e. including grandchildren, and great-grandchildren, etc.).

2.8.3 Example

Example 9. Applying the pattern to Unity software

1. This example invents a novel carousel virtual reality experience. The core object at the centre of the scene spins slowly. Once we've added some more functionality, the script turns all the horse objects in the scene into children of this core object so that they start to spin as well.

Start with a new scene. Add a cube at the origin. This is the core object, and is any non-rotationally symmetric object since we want to be able to see it spin. Add a `RotateObject` C# script containing the following line in its update function, and attach this to the cube. This should make the cube rotate around the vertical axis when the application is run.

```
1     this.transform.Rotate (0.0f, 1.0f, 0.0f);
```

2. Grab a free horse model from an external source. Add it into the scene. You may need to adjust scale. Many externally imported assets are built on millimeter or centimeter scale. Add a horse tag to the horse object (the tag controls are at the top of the Inspector pane).

Add first the horse tag and then add the newly created tag to the horse object). Now scatter a set of the horse objects throughout the scene as shown in Figure 2.8.3.

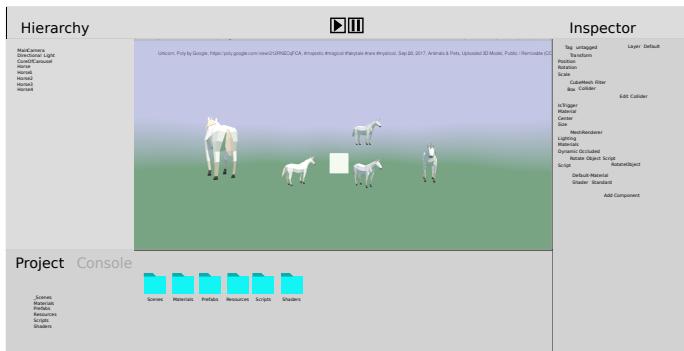


Figure 2.8.3: The scene with a scattering of horse objects, each with the Horse tag (see top of Inspector pane).

3. Update the RotateObject script so that it contains the code shown in Algorithm 9 on the following page. On every update this checks to see if there is a single object with the Horse tag. If there is then it removes the horse tag (so it doesn't get found on the next update) and sets its parent property (part of the transform component) to the object containing the script (the cube). The parent object is referenced using its transform component when creating parent-child links in Unity software. The SetParent function could also be used which gives the option of controlling where the child ends up relative to the parent after the link is established. Setting the parent property to null is used to break an existing link. Having this code in the update method makes for a bit of a slow start, since only one horse is added per update. There is a slight jitter visible when the application starts. However any new horses added while the program is running are automatically added as children when the FindWithTag spots them during an update. An alternative would be to do this in the start function which would require a loop to find all horse tagged objects at that time.

Algorithm 9 Individual horse objects are identified during each update and added as children to the object containing this script component.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class RotateObject : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9          }
10
11     // Update is called once per frame
12     void Update () {
13         this.transform.Rotate (0.0f, 1.0f, 0.0f);
14
15         GameObject horse = GameObject.FindGameObjectWithTag ("Horse"
16             );
17         if (horse != null)
18         {
19             horse.tag = "Untagged";
20             horse.gameObject.transform.parent = this.
21                 gameObject.transform;
22         }
23     }
24 }
```

2.9 Who's the boss: Managing Managers

2.9.1 Description

Objects in a virtual environment can independently sense, reason and act on the virtual world around them. This provides a responsive and reactive environment without a central control process that has to understand and enforce all possible combinations of interactions, allowing possibilities for emergent effects. Occasionally, however, there are scenarios where an element of oversight might be necessary. For example, a collection of similar objects would benefit from a global manager for a coordination role, since such a task is not obviously associated with any single element in the collection.

Consider the example of a traffic simulation consisting of a number of taxis. A user calls for a taxi, and the closest one is dispatched to pick them up. There are potentially ways in which the taxis can negotiate amongst themselves to see who is closest. However contention could arise if two are at exactly the same distance. This process would also require all taxis to communicate with one another, and the user has no obvious point to dispatch their request to.

A taxi manager object would simplify the solution to this problem. The manager would be part of the scene, so that it could share in the computation time allocated to the application, but need not be visible. The clients of the manager would access it only through functions provided in scripts, rather than any direct interaction in the virtual world.

2.9.2 Pattern

The virtual reality manager pattern involves creating a single instance of a manager class and adding this to the scene graph. This is typically included as a component of an invisible object.

The manager class would usually keep track of the

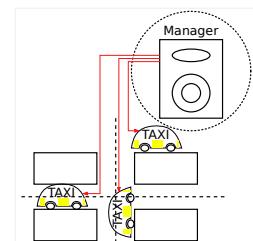


Figure 2.9.1: An invisible manager scene object is used to coordinate groups of objects within the scene.

various entities that it is managing. This is done by either creating all the entities directly as part of the set up function for the manager, or through providing a publicly accessible registration function that each entity can call when they are created by some separate factory-like process.

```
class Manager
{
    private:
        List managedEntities
        function setup ()
        {
            // create and add entities
        }
    public:
        function undertakeAction ()
        {
            ...
        }
        function register (entity)
        {
            managedEntities.add (entity)
        }
}
```

Each client needs to know where to find the manager, as do any entities that need to register themselves with the manager. This is done by manually setting a reference in each client entity object through the virtual reality engine's editor. Alternatively, the manager object might add itself to the scene using a previously agreed name so the clients can search through the scene graph to find it. Another option requires that the manager registers itself with a registry service that the clients can then query to find the manager. Some engines may provide such a registry service but it could also be created by using the Manager pattern to manage managers.

2.9.3 Example

Example 10. Applying the pattern to Unity software

1. This manager pattern is applied to the problem of coordinating traffic lights. We're going to work with a small scale example here since coordinating large sets of objects makes use of concepts covered in other patterns (section 3.6). We'll assume a typical intersection where two roads cross at right angles. We'll need to coordinate the traffic lights representing each of the 4 directions heading out of the intersection.

Each of the individual traffic lights is instantiated from a predefined template that we set up first. Create a traffic light consisting of an empty object, named TrafficLight, as parent to 3 spheres representing the 3 colours of light, and any supporting geometry you desire. The scene structure used is shown in Figure 2.9.2.

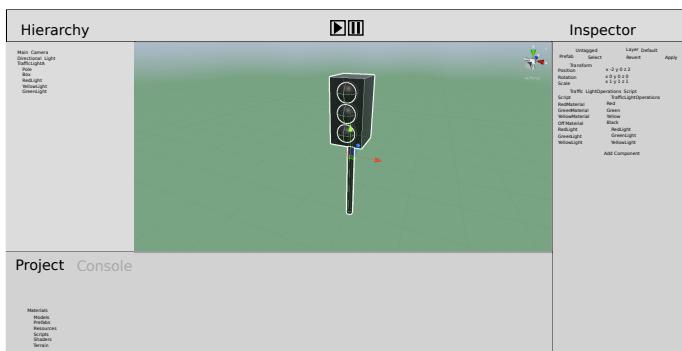


Figure 2.9.2: The key components of the traffic light.

2. We also want to simplify the process of using a traffic light so that the use of the manager pattern does not get overcomplicated with stray details. So we make the traffic light smart enough to manage its own internal operations (changing colour) and provide an interface that the manager can use. This interface is a function called changeColour that the manager can invoke to control a particular traffic light.

Create a C# component called TrafficLightOperations and add it to the parent component for the traffic light. This script needs to be provided with 4 materials, representing the red, yellow, green, and off colours for the light. We include these as public variables so they can be set from the Unity software editor. You need to create these materials and add them to your project assets.

Since traffic lights do not currently use a single colour changing bulb, we also need to identify the individual light within the traffic light that must be changed. We'll add 3 game object public variables so we can easily configure these for any type of traffic light this component may be attached to.

The public interface function SetColour takes in a parameter, which is either "red", "green", or "yellow" and sets up the lights appropriately. There are no safety checks for invalid colours which might be a bad thing for critical systems such as traffic lights. We use the Start function to make a call to SetColour so we can test it, but also to make sure that traffic lights start in a known and safe configuration. The complete script is shown in Algorithm 10 with the setColour function provided in Algorithm 11 on page 74.

3. Test the scene with the single traffic light, and make sure it works.

Then drag the traffic light into the Project's Prefabs folder to turn it into a traffic light template that we can make multiple instances of.

Layout a representation of the intersection, making 4 instances of the traffic light prefab and placing them at the appropriate locations. The resulting scene should resemble that shown in Figure 2.9.3.

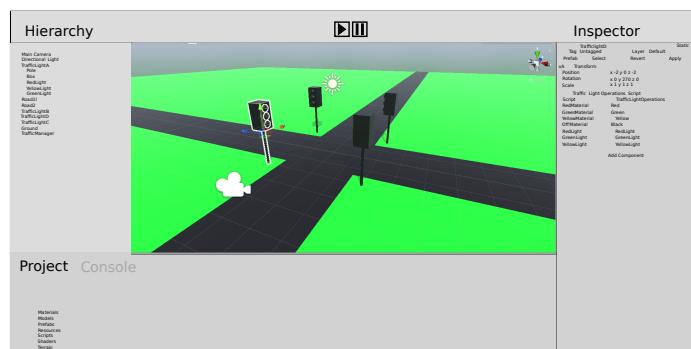
4. The manager pattern can now be applied. Add an empty object to the scene, named TrafficManager. Add a C# script component to this called TrafficManage-

Algorithm 10 Individual traffic lights are able to control their own internal operations and just provide a simple interface that allows them to be operated without the external party needing to understand the structure of a specific traffic light.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TrafficLightOperations : MonoBehaviour {
6
7      public Material redMaterial;
8      public Material greenMaterial;
9      public Material yellowMaterial;
10     public Material offMaterial;
11
12     public GameObject redLight;
13     public GameObject greenLight;
14     public GameObject yellowLight;
15
16     // Use this for initialization
17     void Start () {
18         setColour ("red");
19     }
20
21     // Update is called once per frame
22     void Update () {
23
24     }
25
26     public void setColour (string colour)
27     {
28         // provided separately.
29     }
30 }
```

Figure 2.9.3: A collection of 4 traffic lights whose signals need to be coordinated through the application of the manager pattern.



Algorithm 11 Each individual light is managed according to the desired configuration, provided as a parameter to the function in the manager.

```
1  public void setColour (string colour)
2  {
3      // first switch all lights off
4      redLight.GetComponent<MeshRenderer> ().material =
5          offMaterial;
6      greenLight.GetComponent<MeshRenderer> ().material =
7          offMaterial;
8      yellowLight.GetComponent<MeshRenderer> ().material =
9          offMaterial;
10     // now just switch on the correct colour
11     switch (colour)
12     {
13         case "red":
14             redLight.GetComponent<MeshRenderer> ().material
15                 = redMaterial;
16             break;
17         case "yellow":
18             yellowLight.GetComponent<MeshRenderer> () .
19                 material = yellowMaterial;
20             break;
21         case "redyellow":
22             redLight.GetComponent<MeshRenderer> ().material
23                 = redMaterial;
24             yellowLight.GetComponent<MeshRenderer> () .
25                 material = yellowMaterial;
26             break;
27         case "green":
28             greenLight.GetComponent<MeshRenderer> () .
29                 material = greenMaterial;
30             break;
31     }
32 }
```

ment. The manager needs to know about the 4 traffic lights which we assume are labelled A, B, C and D in a circular ordering. Individual variables are used to keep track of these although ideally a collection would be used in most managers to achieve the same outcome with less code and the ability to scale up to more complex scenarios.

The individual lights are stored in 4 public variables so they can be assigned from the Unity software editor. Once the variables are created remember to drag the parent traffic light objects in to provide their initial value.

5. The traffic manager goes through the following sequence:

- (a) AC - green, BD - red
- (b) AC - yellow, BD - red
- (c) AC - red, BD - red
- (d) AC - red, BD - green
- (e) AC - red, BD - yellow
- (f) AC - red, BD - red

When it gets to the end of the sequence, it starts over again from the beginning.

For simplicity we have each step last 1 second. We use a timing pattern (section 5.2 on page 173) to switch configurations. The complete script for the TrafficManager is shown in Algorithms 12 and 13. This function is a lot longer than it really needs to be and can be improved through abstraction (section 3.5 on page 104) and the use of the state machine pattern (section 6.1 on page 205).

2.10 Exercises

Exercise 11. The virtual reality space invaders experience features flying saucers that descend from the sky. Complete the following challenges:

Algorithm 12 The traffic manager cycles through different values for all of the lights that it controls, waiting 1 second between changes (part 1).

```
1  ...
2  public class TrafficManagement : MonoBehaviour {
3
4      public GameObject lightA;
5      public GameObject lightB;
6      public GameObject lightC;
7      public GameObject lightD;
8
9      // Use this for initialization
10     void Start () {
11         StartCoroutine (cycleLights ());
12     }
13     IEnumerator cycleLights() {
14         while (true)
15         {
16             lightA.GetComponent <TrafficLightOperations> () .
17                 setColour ("green");
18             lightC.GetComponent <TrafficLightOperations> () .
19                 setColour ("green");
20             lightB.GetComponent <TrafficLightOperations> () .
21                 setColour ("red");
22             lightD.GetComponent <TrafficLightOperations> () .
23                 setColour ("red");
24             yield return new WaitForSeconds (1);
25             lightA.GetComponent <TrafficLightOperations> () .
26                 setColour ("yellow");
27             lightC.GetComponent <TrafficLightOperations> () .
28                 setColour ("yellow");
29             lightB.GetComponent <TrafficLightOperations> () .
30                 setColour ("red");
31             lightD.GetComponent <TrafficLightOperations> () .
32                 setColour ("red");
33             yield return new WaitForSeconds (1);
34             // continued ...
```

Algorithm 13 The traffic manager cycles through different values for all of the lights that it controls, waiting 1 second between changes (part 2).

```
1      // continuation ...
2      lightA.GetComponent <TrafficLightOperations> () .
3          setColour ("red");
4      lightC.GetComponent <TrafficLightOperations> () .
5          setColour ("red");
6      lightB.GetComponent <TrafficLightOperations> () .
7          setColour ("green");
8      lightD.GetComponent <TrafficLightOperations> () .
9          setColour ("green");
10     yield return new WaitForSeconds (1);
11     lightA.GetComponent <TrafficLightOperations> () .
12         setColour ("red");
13     lightC.GetComponent <TrafficLightOperations> () .
14         setColour ("red");
15     lightB.GetComponent <TrafficLightOperations> () .
16         setColour ("yellow");
17     lightD.GetComponent <TrafficLightOperations> () .
18         setColour ("yellow");
19 }
```

1. Create a template for one flying saucer.
2. Using a script, instantiate 3 flying saucers at different positions in the sky when the application starts.
3. Using a script, ensure that each flying saucers rotates continuously about the vertical axis.
4. Using a script, ensure that each flying saucer descends slowly towards the ground.

3 *Algorithmic Twiddles*

3.1 All The Programming Skills You Will Ever Need

Becoming an accomplished programmer takes more than the brief introduction covered in this section. However many small behavioural elements in VR applications can be constructed using a small set of instructions applied using the patterns covered in this book, and a clear understanding of how your design is to be turned into (virtual) reality.

There are a small number of programming statements that are commonly used and which are worth being familiar with. These include:

assignment: requires the concept of a variable (which stores information) and the transfer of information between different variables. Properties in the scene graph are examples of variables.

conditional: how to make a decision about which operations to do next, based on current information values stored in variables.

iteration: how to repeat things. how to repeat things.

abstraction: programs become complicated very quickly unless we learn to recognize the common and core

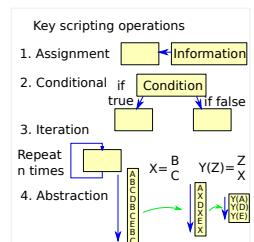


Figure 3.1.1: Typical operations performed during scripting elements of a virtual reality application.

steps of any process and abstract these out as sequences of operations that can be reused. This also provides meaningful names to these sequences and allows our programs to be more easily readable, and to more closely match with a human understanding of how things work (and helps humans to create programs in the first place).

The particular statements used are grouped into blocks of code, delimited by a block boundary. Block boundaries in many languages use the brace or curly bracket symbols {} to indicate the start and end of blocks. Within a block statements are executed one at a time from top to bottom. The order in which blocks are executed in a virtual reality system depends on how they are used, such as being invoked as part of a function or triggered by an event in the virtual world.

3.2 *Variable Assignment*

3.2.1 *Description*

Your programs store the information representing various properties in the scene, as well as any additional information used to support any custom operations required for your application.

Information is stored in variables. You can visualize this as a box into which you can place information. This box is stored in a portion of the memory of the computer that is allocated for your use. The length of time that this box is allocated to you is defined by scope rules.

Each particular box has its own name, so it can be uniquely identified. Visualize the name written on the top of the box.

The box stores information. There are different types of information: numbers, text, colours, vectors, geometric shapes, sounds, images to name a few. All of these are converted to information when they are assigned to the variable (stored in the box). Visualize this by seeing your

information turned to mushy gray goop as it is placed in the box.

We recover our information by interpreting it according to the type of information that it is. Visualize a description of the type being written on the side of the box. Visualize the mushy gray goop being poured through a sieve of a particular type that miraculously reconstructs the original information you assigned to (stored in) the variable.

3.2.2 Pattern

Purely as examples, the types of variables you are likely to encounter include:

primitive type: These are fixed by the programming language you are using. Examples: `int` (whole numbers), `float` (allows a decimal point and fractional part), `string` (sequence of text symbols).

engine specific type: These are common types of information used by virtual reality applications and have conveniently been defined by the VR engine. Examples: `Vector` (for 3 spatial coordinates), `Texture` (for image content).

user defined type: You get to make your own types as well. Normally these would be collections of associated information. For example, a `Player` type might be composed from a `string` type (to represent the name of the player) and a `Vector` type (to represent the position of the player).

When working with variables you need to first declare and initialize them. This involves stating: the type of the variable, the name of the variable and the initial (starting) value of the variable. Visualize this as creating the box, writing the variable name on the top, the type on the side and plonking the information's gray goop inside.

```
variable_type variable_name = initial_value
```

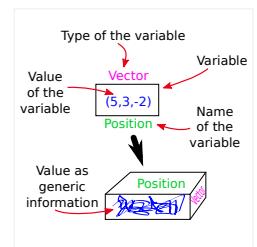


Figure 3.2.1: A suggested mental model for understanding the representation of information in the form of variables.

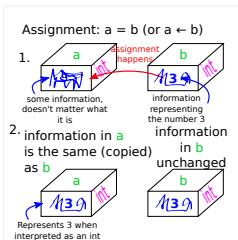


Figure 3.2.2: A visual metaphor for the changes that occur during the assignment operation.

Providing a new value to the variable is done with an assignment operation:

```
variable_name = new_value
```

Think of this as $\text{variable_name} \leftarrow \text{new_value}$ to get an idea of the direction information is moved (the equals sign is very misleading but will have to do until we have keyboards that allow an \leftarrow to be easily typed in). If new_value is a variable then the information (gray goop) is taken out of its box (copied, the original is left behind) and poured into the box for variable_name . The previous goop in variable_name is lost in the process (overwritten).

The new_value can also be a number or text sequence (for appropriate variable_types), or an arithmetic expression involving several variables or constant values.

The types of information on each side of the assignment operator ($=$) should ideally be the same. If they differ then your VR engine may apply an arcane translation process to convert the information from the type of new_value to the type of variable_name . The exact nature of this process may depend on the language and system being used but in general works predictably (at least to the casual reader of your program) only in a small set of cases.

Variables that are not a primitive type but that refer to objects (instances of classes) are of a primitive type called a reference. The object itself is stored in a separate unnamed block of memory, and the reference contains just an arrow identifying that block. This is a subtle distinction but does explain apparent inconsistencies in the rules about parameter passing in functions (section 3.5 on page 104), and concepts such as scope and lifetime (section 3.7 on page 126).

3.2.3 Example

Example 12. Applying the pattern to Unity software

1. This example shows off some of the variable types and addressing conventions when using C# within Unity software.

Start off with an empty project and a new scene. Add a new object to the scene. It doesn't matter what this is since we just use it to attach a script component which demonstrates some examples of variables and assignment. Create a new C# component in the Project window called VariableDemo. Drag this component onto the object in the scene that you've just created. Double click on the VariableDemo component in the Project pane to open it up in the editor of the development environment.

We'll add all the code in the body of the Start function so that it runs as soon as the application is started.

2. First we create some variables using primitive types. Primitive types are built into the specification of the language, and represent the smallest quantum of information that the program can work with. They are directly copied and assigned to other variables of the same type.

The code below creates variables using some of the primitive types commonly used in virtual reality applications:

```
1 int varThatIsInt = 7;  
2 float timeCounter = 37.4f;  
3 bool onNotOff = true;
```

This script reveals some conventions that are used. Variables names use camelCase; starting each new word with a uppercase letter. The convention used here is to start variable and function names with a lower case letter to distinguish them from new types (classes) that we can also create.

Variables are initialized by assigning them a value when they are declared, as shown here. Integers are whole numbers, floats can have a decimal point, and

booleans can be either true or false. Float constants end in an *f* to distinguish them from the primitive type: double.

Save the file containing the script. Check that the Console window in Unity software is not reporting any error. Any console messages that are shown indicate the line where the problem is first noticed. Check that lines and the ones around it (including before the problem line) in case you have transcribed any portion of the code incorrectly.

3. Running the application with this code shows no visible change. Other patterns explore ways to watch the internal progress of a program (section 4.3 on page 150) but for the moment we make use of a simpler pattern (section 4.2 on page 145) to confirm that our mental model of what is happening matches with the evidence that is produced by the virtual reality engine.

Include these statements immediately below the ones previously added.

```
1 print ("Integer variable value: " + varThatIsInt);
2 print ("TimeCounter value: " + timeCounter);
3 print ("Boolean variable's value: " + onNotOff);
```

The print function reports its argument in the Console window of the Unity software editor. This is not terribly useful to a user of a virtual reality application but is of incredible significance to a developer of such applications who wants some evidence of what the program is actually doing.

The print function expects a parameter of type string. Variables of other types are converted (cast) to their string equivalent and combined with any other strings in the parameter list using the + operator. This operator which performs addition in ints and floats performs concatenation when applied to strings.

The program should now produce the output shown in Figure 3.2.3 on the next page once the application is

run by pressing the play button in the Unity software editor. The values associated with each variable are retrieved, converted to strings, combined with the other text and shown in the console. The labels provided as the first part of the parameter to the print function are not required but are a good habit to get into to help clearly identify which print statement is responsible for any particular output.

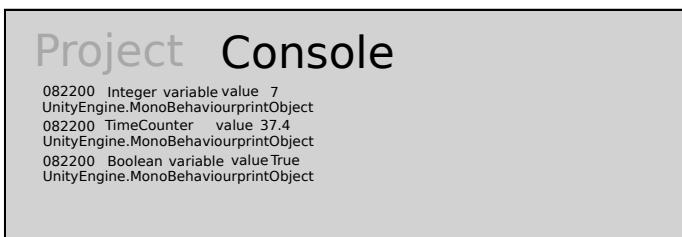


Figure 3.2.3: The print function displays the value of the expression provided as its parameter.

- Unity software provides various types that are used to represent quantities commonly used in virtual reality applications. Examples include the Vector3 which represents x, y and z values associated with positions and directions, and Color which represents the amount of red, green and blue light associated with a particular colour.

Include the following code in body of the Start function:

```

1 Vector3 myDirection = new Vector3 (1.0f, 1.5f, -0.3
f);
2 Color red = new Color (1, 0, 0);
3 print ("Direction Y: " + myDirection.y);
4 print ("Colour bits: " + red);

```

Non-primitive types are reference types which means that the variable itself contains a reference (pointer) to a separate storage box containing the actual object. The new operator is required to create this storage box and to retrieve the reference to it. The new operator needs the name of the type and any initial values required to construct and populate the storage box.

Compound types of this kind are an aggregation of multiple values. Each of these fields has its own name, specific to the definition of the compound type. For example, Vector3 has fields x, y and z. Color has fields r, g, b and a. Each field is its own variable, identified using the name of the variable, followed by a “.”, and followed by the name of the field (e.g. myDirection.y). Compound objects normally define how they are converted to a string representation so that they can be used in a print function as shown when the variable of type Color is printed.

5. It is possible to define your own compound types out of other types. The code in Algorithm 14 on the facing page constructs the basic elements of a type to represent a ray (section 5.5 on page 194) which consists of an origin and direction field, each of which is a Vector3. Two functions are part of this definition; the constructor used to assign values to the fields when a new value of this type is created, and the string conversion function ToString used to provide a meaningful string representation for use in the print statement.

Example 13. Applying the pattern to Unity software

1. The other significant aspect of being able to represent and store information using variables is that the assignment operation can transfer not only a constant value or the value of another variable, but assign the result of a much more complex expression. This allows the nature of the information in the program to be transformed into other forms that might be more directly applicable to specific aspects of the virtual reality application.

Create a new Unity software project, and add an arbitrary object to the scene. Create a new Project asset as a C# script called InformationTransformation and attach that to the object in the scene.

Algorithm 14 A code sample to demonstrate declaring and initializing variables of various primitive, engine defined and developer defined types.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class VariableDemo : MonoBehaviour {
6
7      class Ray
8      {
9          Vector3 origin;
10         Vector3 direction;
11
12         public Ray (Vector3 o, Vector3 d)
13         {
14             origin = o;
15             direction = d;
16         }
17         public override string ToString ()
18         {
19             return "[" + origin + "->" + direction + "]";
20         }
21     }
22
23     // Use this for initialization
24     void Start () {
25         int varThatIsInt = 7;
26         float timeCounter = 37.4f;
27         bool onNotOff = true;
28         print ("Integer variable: " + varThatIsInt);
29         print ("TimeCounter value: " + timeCounter);
30         print ("Boolean variable's value: " + onNotOff);
31
32         Vector3 myDirection = new Vector3 (1.0f, 1.5f,
33                                         -0.3f);
34         Color red = new Color (1, 0, 0);
35         print ("Direction Y: " + myDirection.y);
36         print ("Colour bits: " + red);
37
38         Ray pointerDirection = new Ray (new Vector3
39                                         (0,0,0), new Vector3 (1,0,0));
40         print ("Ray value: " + pointerDirection);
41     }
42
43     // Update is called once per frame
44     void Update () {
45     }
```

2. Add the following code to the body of the Update function.

```
1 Vector3 updatedPosition = this.gameObject.transform  
    .position + new Vector3 (0.01f, 0, 0);  
2 this.gameObject.transform.position =  
    updatedPosition;
```

When you run the application you should see the object move sideways across the screen. This code is creating a new variable named updatedPosition of type Vector3. During initialization the variable is assigned the value resulting from an expression. This expression adds the Vector3 value representing the current position of the object to another Vector3 value representing a small increment in the x (sideways) direction. The current position is a field in:

- the position field in the transform variable which is
- the transform field in the gameObject variable which is
- the gameObject field in variable this which is
- the C# variable representing the current object (script itself, an object of the InformationTransformation class).

The resulting value of this expression is a new position, a small distance to one side of the current position. The next line overwrites the current position with this new value. This change of the information in the scene graph effectively puts the object in a different position. The process is repeated each time the display is updated, resulting in the apparently continuous movement of the object.

3. We can actually achieve the same result in one step by directly assigning the value of the expression straight into the variable this.gameObject.transform.position. It doesn't matter that the same variable is both the source and destination of the assignment operation.

The expression is evaluated first, and then the result is written into the variable.

```
1  this.gameObject.transform.position = this.  
    gameObject.transform.position + new Vector3  
    (0.01f, 0, 0);
```

This operation (incrementing a value by a given amount) is so common that there is a shorthand notation, combining the assignment and addition operation.

```
1  this.gameObject.transform.position += new Vector3  
    (0.01f, 0, 0);
```

4. A range of arithmetic operations are used in the expressions. This example produces circular motion using the equation for a point of a circle:

$$\begin{aligned}x &= r \sin(\theta) \\y &= r \cos(\theta)\end{aligned}$$

Here the radius of the circle is represented with r , and the angular position of the point in the circle is represented by θ . If we use a counter or time related variable for the angle, then the coordinates resulting from this expression trace out a sequence of points around the circumference of a circle. The following script demonstrates this process:

```
1  float radius = 3.0f;  
2  float circlex = radius * Mathf.Sin (Time.time);  
3  float circley = radius * Mathf.Cos (Time.time);  
4  this.gameObject.transform.position = new Vector3 (  
    circlex, circley, 0);
```

This could also be simplified into fewer lines of code but you need to decide if the meaning is still clear.

```
1  this.gameObject.transform.position = new Vector3  
    (3.0f * Mathf.Sin (Time.time), 3.0f * Mathf.Cos  
    (Time.time), 0);
```

3.3 Conditionals

3.3.1 Description

Some actions in a virtual reality application are conditional, and are only appropriate if a particular condition is true. For example, an object may move downwards but only if the resulting position is not below ground level. The height of the object needs to be compared to the height of the ground to determine whether the condition for movement has been satisfied.

Conditions are expressions that evaluate to either true or false. Most computing languages support a Boolean type which allows variables of this type to only assume one of these two values (true or false). Various operators exist that evaluate an expression and return a Boolean value, including:

- equality: symbol `==`. The expression $a == b$ is true only if a and b have the same value.
- inequality: symbol `!=` or `<>`. The expression $a != b$ is true if a and b have different values.
- comparison: symbol `>`, `>=`, `<`, `<=`. The expression $a < b$ is true if a and b are both numeric types and a is less than b .
- conjunction: symbol `&&`, *and*. The expression $a \&\& b$ is true if both a and b are Boolean values or expressions, and both a and b are true.
- disjunction: symbol `||`, *or*. The expression $a || b$ is true if both a and b are Boolean values or expressions, and either a or b are true.
- negation: symbol `!`, *not*. The expression $!e$ is true if the Boolean expression e is false, and false if e is true.

Common issues that arise through the use of conditions involve confusing:

- = and ==. The symbol = is the assignment operator. This operator has the side effect of returning the value being assigned, and so using this in a Boolean expression may actually produce a valid statement albeit not the one you may have intended. The symbol == is the equality operator. Some languages may also not warn you if the equality operator is accidentally used when an assignment operation is intended. The program happily compares two values and discards the result without performing any assignment at all.
- && and & (similarly with || and |). The symbol && is the Boolean operator to check that both arguments are true at the same time. The symbol & operates on the binary representation of numeric values. In particular circumstances they can appear to behave in the same way when tested with simple data (particularly 1s and 0s) so the effect of the difference may only become apparent much later.

3.3.2 Pattern

A conditional is included into a script using a pattern of the form:

```
if (condition)
{
    program statements to run if the
        condition is true
}
else
{
    program statements to run if the
        condition is false
}
```

The condition is any expression that evaluates to a boolean value. Some languages also allow numeric expressions where it is assumed that 0 corresponds to false, and any other value corresponds to true.

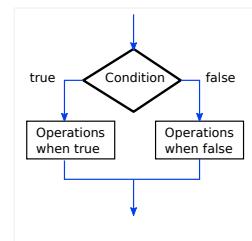


Figure 3.3.1: The conditional statement allows different program statements to be run depending on the value of an expression.

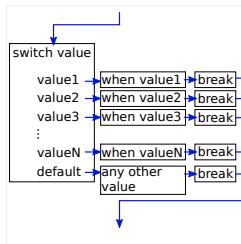


Figure 3.3.2: The switch statement extends the power of the conditional by designating different blocks of code corresponding to each designated value of a variable.

The else clause portion of the pattern is optional, and is left out if not required.

An alternative form of the conditional is the switch statement. This allow individual responses to a number of different values associated with an expression. The typical form of the switch statement is:

```

switch (expression)

  case value1:
  {

    program statements to run if
      the expression == value1

  }
  break

  case value2:
  {

    program statements to run if
      the expression == value2

  }
  break

  case value3:
  {

    program statements to run if
      the expression == value3

  }
  break

  ...
  default:
  {

    program statements to run in
      any other situation

  }
  break
  
```

The switch statement evaluates the expression provided. If the result matches the value provided with the first case then it runs the script provided for that case. The

check is then repeated for the remaining cases. If none of them succeed, and a default case is provided then the script for the default case is run.

The break at the end of each case forces an exit from the switch statement at the point. This is crucial for ensuring that one, and only one, case is used. If the break is not present, then the script for the next case is also run regardless of whether the value for that case matches or not. There are cases when this behaviour might be desirable but many more where it is not. The recommended strategy is always to include the breaks, even in cases such as the last case where they serve no role. Explicitly flag in the comments where you've left them out deliberately. It is easy to modify a switch statement to add in more options and accidentally end up with strange behaviour that is hard to diagnose otherwise.

3.3.3 Example

Example 14. Applying the pattern to Unity software

1. In this example we create virtual barriers for objects in a virtual world. In this specific case we enable the physics subsystem in the virtual reality engine so that objects experience virtual gravity and virtual forces. We do not enable the collision detection and response (section 7.1 on page 247) that would normally provide obstacles. Instead the barriers are invisible and controlled through the use of the conditional pattern.
2. Create a new Unity software project. Add an object to the scene. This is the particle affected by the virtual physics. In the Inspector pane, use the Add Component button to add a Physics/Rigidbody component to this object. If you run the application at this stage, the particle should drop down under the influence of the virtual gravity force and disappear off the bottom of the screen. Increase the y coordinate of the object so it has some distance to drop before it hits the y=0 level. An example is shown in Figure 3.3.3 on the next page.

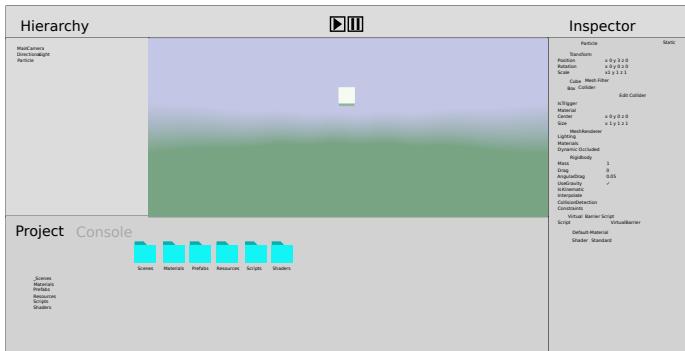


Figure 3.3.3: A single particle with a Rigidbody component attached so that it is subject to the physics simulation.

Algorithm 15 A conditional that creates a virtual barrier at height o . If an object drops below this height, it is reset back to the o level and given an upwards push.

```

1  if (this.gameObject.transform.position.y < 0.0f)
2  {
3      this.gameObject.transform.position = new Vector3 (
                this.gameObject.transform.position.x, 0.0f,
                this.gameObject.transform.position.z);
4      this.gameObject.GetComponent <Rigidbody> ().velocity
            = new Vector3 (0, 10.0f, 0);
5 }
```

3. Create a new C# component and add it to this object. Call this component VirtualBarrier. Add the code shown in Algorithm 15 to the body of the Update function. The conditional evaluates an expression that compares the height of the object (the y component of the position) to o . If the height is below o (the object has dropped below the barrier) the code in the body of the conditional is executed. This resets the height of the object back to o (not essential but a way of making the code more robust) and gives the object an upwards velocity so it now rises about the barrier.

The condition could be phrased as `this.gameObject.transform.position.y == 0.0f`, but it is a bad idea to compare floating point values for exact equality. Rounding and quantization errors can creep in to prevent equality being achieved. More

significantly the physics simulation is likely to jump in discrete steps and so reaching any predefined height exactly is nearly impossible.

Try this out and watch the object bounce.

4. For more fun:

- (a) Place a plane at the $y=0$ level so it looks like the object is actually bouncing off something.
- (b) Create more virtual barriers. Surrounding the object with a box (6 virtual barriers) could produce some interesting behaviour.
- (c) Include some randomness (section 6.3 on page 237) to the velocity when the object hits a barrier. This allows it to bounce in different directions, and perhaps hit one of the other barriers.
- (d) Turn the object into a prefab, and then add lots of the prefabs to the scene at different positions. This should create a virtual world with some interesting activity.

3.4 Iteration

3.4.1 Description

The value in writing programs is that it allows us to automate repetitive tasks and to sit back and relax while the computer does all our work for us.¹ There are a number of programming constructs that repeat a block of instructions a given number of times, or until some condition is satisfied. We mainly use a pattern for a fixed number of repetitions in our virtual reality applications. Iterating in a virtual reality application can stop the application from responding to user interaction until the iteration is complete and so it is advisable to confine long loops to start up phases of the application. We may also use iteration patterns for repeating a process for each element in a list.

¹ There are some other benefits as well which ideally become clear as we progress through this book.

Virtual reality applications typically already have a standard form of iteration built into them in the form of the game loop (section 2.7 on page 59). This allows the frame event handler to be invoked over and over again on every frame (30 to 100 times per second) and it is worth considering whether any required repetition is already provided by this.

3.4.2 Pattern

This is a pattern for a loop that repeats exactly n times. We use the variable i as a loop counter and this takes on the values $0, 1, 2, \dots n - 1$ in each successive iteration of the body of the loop. The convention of starting to count from 0 is peculiar to computer programming but does support other idiosyncrasies such as referring to the first element in a list as element 0.

for ($i = 0; i < n; i++$)

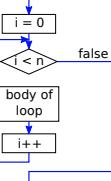


Figure 3.4.1: The for loop supports iteration that occurs a fixed number of times, including a version that repeats exactly n times.

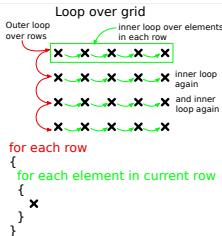


Figure 3.4.2: The nested for loop supports iterating over complex structures since the entire inner loop is repeated on every iteration of the outer loop.

```

for (i = 0; i < n; i++)
{
    // Body of the loop.
    // Any statements here are repeated
    // exactly  $n$  times.
    // The variable  $i$  will have values 0, 1,
    // 2, ...,  $n - 1$  corresponding to which
    // iteration of the loop is currently
    // active.
}
  
```

Iterating over multidimensional structures such as grids require a loop to iterate over every row, and for each row, an additional loop to iterate over every column. Each iteration of the row loop requires a complete repetition of the entire column loop. This is achieved by placing the column loop in the body of the row loop, a process described as nesting.

```

for (i = 0; i < number of rows; i++)
{
  
```

```

for (j = 0; j < number of columns; j++)
{
    // do something with row i, column j
}

```

We have to use two different names for our loop counters. Loop counters can have any legal variable name and should be given meaningful names if their value is being used for a particular purpose. Historically anonymous loop counters have been named i, j, k, \dots .

Do not do this:

```

for (i = 0; i < number of rows; i++)
{
}
for (j = 0; j < number of columns; j++)
{
    // do something with row i, column j
}

```

This variation only covers iterates over a single column and a single row. Even worse, variables i and j do not always have meaningful values inside the other loop, assuming the programming environment even allowed this to run. Make sure you understand the difference between the good and the bad pattern as this will confirm that you appreciate the role of the iteration pattern.

3.4.3 Example

Example 15. Applying the pattern to Unity software

1. This example shows the use of the iteration pattern for procedurally populating a scene with multiple instances of a particular object template (a prefab using Unity software terminology). We could lay it out in a grid by directly using the nested for loop pattern

but have instead decided on something different: a spiral.

A spiral is produced by laying out objects in a circular pattern but increasing the radius after each complete circle. As a reminder, the coordinates of points on a circle are given by the expression:

$$\begin{aligned}x &= r \sin(\theta) \\y &= r \cos(\theta)\end{aligned}$$

If we regularly increase the angle θ using our iteration pattern then we can get points progressively positioned around the circumference of a circle.

2. This example is also going to demonstrate a few other patterns that represent typical virtual reality application patterns. We are going to make extensive use of public variables, as these are parameters that are set each time the script is used so that the script is reusable in multiple different scenarios. The specific scenario that this example demonstrates is the creation of a spiral arrangement of rocks, in a tribute to the traditional henge. However the code makes no reference to rocks, or specific shapes, so this script can be reused for other spiral placement purposes.

The parameters we use are:

- The prefab for the object that is to be placed. When we use the script, we assign a rock prefab using an externally created model.
- The total number of objects that must be placed. This is the upper bound for the for loop that we create.
- The number of objects that are placed in a single complete rotation of the spiral. This helps us work out the angular change from one object to the next.
- The starting radius of the spiral (radius of the inner circle) in virtual metres, and the change in radius

from one rotation of the spiral to the next, also in virtual metres.

3. We use the manager pattern (section 2.9) to handle the task of creating the individual rocks in the spiral. Start with a blank scene, and add an empty object named RockManager. Create a new prefab object to represent a rock. Create a C# script named SpiralPlacement, and include the program code shown in Algorithm 16.

For clarity we compute a few values that are used on each iteration of the loop. This includes *anglePerStep* representing the amount that the angle has to increase by for every value of the loop counter, *i*. We also keep a local variable *radius* which we increase in each step of the loop. The increment (*radiusChangePerStep*) is calculated so that the radius value has increased by the *radiusIncreasePerRotation* value after having made a complete circle.

The iteration itself is done in the for loop. The variable *i* starts at 0 and counts up to *totalNumberOfObjects* – 1. Thus the loop repeats its body exactly *totalNumberOfObjects* times, once for each object it creates. After computing the position on the spiral it instantiates the object prefab at that position. The radius counter is also increased on each iteration of the loop.

For neatness, we make any new objects instantiated children of the manager object. This helps keep the scene graph manageable if any debugging needs to occur.

4. In this script we are effectively using three synchronized counters. The variable *i* is the loop counter while the angular position and radius are either calculated directly from *i*, or incremented separately as part of the loop. It is possible to manage them all in the same for loop, although doing so involves a loop instruction that is quite complex to read. This equivalent alternative is shown in Algorithm 17.

Algorithm 16 Iterative placement of objects in a spiral pattern whose properties are determined by the public properties that this component provides.

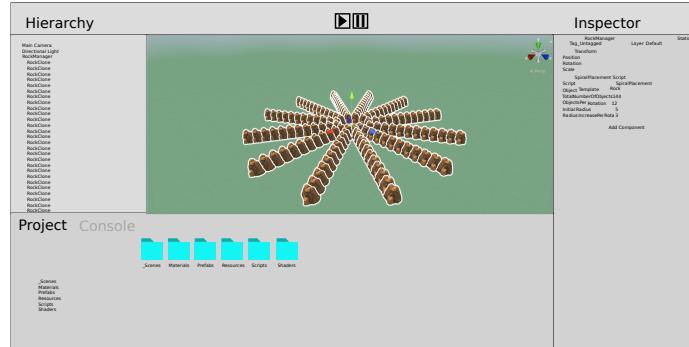
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class SpiralPlacement : MonoBehaviour {
6
7      public GameObject objectTemplate;
8
9      public int totalNumberOfObjects = 100;
10     public int objectsPerRotation = 20;
11     public float initialRadius = 5.0f;
12     public float radiusIncreasePerRotation = 3.0f;
13
14     // Use this for initialization
15     void Start () {
16         float anglePerStep = 2 * Mathf.PI /
17             objectsPerRotation;
18         float radius = initialRadius;
19         float radiusChangePerStep =
20             radiusIncreasePerRotation /
21             objectsPerRotation;
22
23         for (int i = 0; i < totalNumberOfObjects; i++)
24         {
25             float x = radius * Mathf.Sin (i * anglePerStep);
26             float y = radius * Mathf.Cos (i * anglePerStep);
27
28             GameObject instance = Instantiate (
29                 objectTemplate, new Vector3 (x, 0, y),
30                 Quaternion.identity);
31             instance.transform.SetParent (this.transform);
32
33             radius = radius + radiusChangePerStep;
34         }
35     }
36 }
```

Algorithm 17 A variation on the iterator pattern for a for loop that updates three synchronized quantities. The number of iterations is still only determined by the totalNumberOfObjects variable.

```
1 void Start () {
2     float anglePerStep = 2 * Mathf.PI /
3         objectsPerRotation;
4     float radiusChangePerStep =
5         radiusIncreasePerRotation / objectsPerRotation;
6     int i;
7     float radius;
8     float angle;
9     for (i = 0, radius = initialRadius, angle = 0;
10         i < totalNumberOfObjects;
11         i++, radius += radiusChangePerStep, angle +=
12             anglePerStep)
13     {
14         float x = radius * Mathf.Sin (angle);
15         float y = radius * Mathf.Cos (angle);
16         GameObject instance = Instantiate (objectTemplate,
17             new Vector3 (x, 0, y), Quaternion.identity);
18         instance.transform.SetParent (this.transform);
19     }
}
```

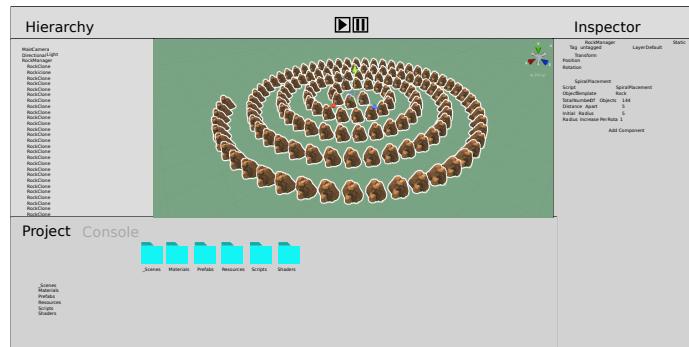
5. The output of this code is shown in Figure 3.4.3. Since the same number of objects are placed in each ring of the spiral they do get further apart as they are placed further from the centre.

Figure 3.4.3: A rock spiral created by iteratively instantiating the rock prefab and placing at a particular position computed from the loop counter.



6. As an alternative we can adjust the amount by which angle and radius are changed for each iteration of the loop to space the objects equal distances apart. In this case, the angular distance decreases as the radius increases. The code for this variation, using the same loop structure, is shown in Algorithm 18, and produces the results shown in Figure 3.4.4. The key

Figure 3.4.4: The rock spiral with objects at equal distances.



change is to calculate the *anglePerStep* variable so that it corresponds to a constant offset between objects, as defined by the variable *distanceApart*. The benefit of working with angles in radians is that the arc length

Algorithm 18 The increments for angle and radius are modified inside the loop to ensure that objects are spaced at equal distances.

```
1  public class SpiralPlacement : MonoBehaviour {
2
3      public GameObject objectTemplate;
4
5      public int totalNumberOfObjects = 100;
6      public float distanceApart = 5.0f;
7      public float initialRadius = 5.0f;
8      public float radiusIncreasePerRotation = 3.0f;
9
10     // Use this for initialization
11     void Start () {
12         float anglePerStep;
13         float radiusChangePerStep;
14
15         int i;
16         float radius;
17         float angle;
18         for (i = 0, radius = initialRadius, angle = 0;
19              i < totalNumberOfObjects;
20              i++, radius += radiusChangePerStep, angle +=
21                  anglePerStep)
22         {
23             anglePerStep = distanceApart / radius;
24             radiusChangePerStep = radiusIncreasePerRotation
25                 * anglePerStep / 2.0f * Mathf.PI;
26             float x = radius * Mathf.Sin (angle);
27             float y = radius * Mathf.Cos (angle);
28             GameObject instance = Instantiate (
29                 objectTemplate, new Vector3 (x, 0, y),
30                 Quaternion.identity);
31             instance.transform.SetParent (this.transform);
32         }
33     }
34 }
```

around the segment of the circle is the product of the radius with the angle subtended by that arc. The change in radius is the proportion of *radiusIncreasePerRotation* in the ratio of *anglePerStep* : 2π .

3.5 Abstraction

3.5.1 Description

Managing the complexity of the virtual reality applications that you develop is fundamental to producing a workable design that can actually be implemented and that can continue to be extended during its working life. Good programmers are not those that can generate complicated systems but rather those that can abstract away the complexity by working with designs that decompose the system into the interoperation of simple and well defined objects. The object-oriented paradigm in section 2.2 on page 25 represents one strategy to achieve abstraction. Much of the object-oriented decomposition is already provided by the virtual reality engine and we need to embrace and utilize that design. We can also further incorporate abstraction into our algorithms and scripts by abstracting key operations to individual functions.

Functions are portions of the program code that represent a common operation, or duplicated sequence of steps, that occurs in multiple locations in the program code. Rather than rewriting the same set of instructions over and over again, we isolate them out to a single function, and just call (invoke) that function each time we need it. Identifying opportunities for functions is normally done while designing the application, but we can also do so while maintaining the code of an application. If we see the same operations repeated we can factor them out to a single function.

Since each occurrence of the common code may involve minor variations we have the opportunity to pass

parameters to the function. These are variables that have the specific values that may differ between each occurrence. Functions also have the ability to return the results of any computation that they do.

It does help to provide a meaningful name for each function, related to what it actually does. A region of code that would have been long and difficult to read can then be factorized out into a sequence of function calls that are then readable as the set of steps involved in achieving the outcome.

3.5.2 Pattern

The pattern for function is:

```
// Some comments describing the function,
// what it does and the nature of the
// parameters.
function nameOfFunction (Type parameter1,
    Type parameter2, ..., Type parameterN)
{
    // sequence of program statements.
    return resultVariable
}
```

The name provided for the function should relate to the purpose of the function. The function becomes even more readable if comments are included to describe its purpose. The list of parameter values represents the information that is provided to the function. These are variables that are available for use in the code making up the body of the function. These variables are discarded when the function ends. Each parameter can have its own type and name mirroring the conventions used when we declare variables.

During the execution of the function we might create a variable that holds of the result of the computation. The value of this variable is provided as the result of the function by using the return statement.

Invoking the function is done as part of any expression, with the return value of the function being the value that is incorporated into the expression. The pattern below shows a simple but standard scenario where the function is invoked, and its returned value assigned to a variable.

```
myVariable = nameOfFunction (3, 17.0,
                            "hello world")
```

The values provided to the function when it is invoked are copied to the function parameter variables by matching position of values in the argument list with the position of parameter in the function's parameter list. Thus the number 3 is assigned to parameter1, the value 17.0 is assigned to parameter2, and the string "hello world" is assigned to parameter3. The type of each parameter must agree with the value that is passed to it.

We can also pass variables as parameters to a function. In this case, the value in the variable is copied into (assigned to) the parameter variable. Changes to the parameter variable in the function do not affect the value in the variable used when the function is invoked. This reasoning is a bit harder to understand when passing objects as parameters; a reference to the same object is copied and so properties of the object that are changed are then visible when viewing the object via the original reference.

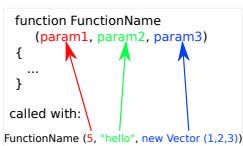


Figure 3.5.1: The arguments provided to a function when it is invoked are copied to the parameters in the corresponding positions.

3.5.3 Example

Example 16. Applying the pattern to Unity software

1. We'll demonstrate this pattern by developing a flocking simulation in Unity software. Each object in a flock needs to do some simple things involving checking its nearest neighbours and:
 - (a) Aligning its direction towards average position of the neighbours, or away from this position if we're too close.

- (b) Aligning its direction with the average heading of its neighbours.
2. First we'll set up the scene. Create a new Unity software scene, and add a background plane to represent the sky (and the allowed region of movement). Rotate this around the x-axis by -90 degrees to form the backdrop for the boids. The default plane has bounds of -5 to 5.
- Create a generic boid object with some distinctive material. Attach a C# script component called Boid-Movement. Convert the Boid object into a prefab. Add a few instances of this prefab to the scene.
3. Now we apply the process of abstraction by decomposing the flocking problem using a top-down approach. During the update step, each boid needs to:

- (a) Identify its neighbours
- (b) Select a direction based on neighbours
- (c) Ensure that direction does remains within the bounds of movement
- (d) Move a step in that direction

Each of these steps is represented using a function. Information returned by one function (such as the list of neighbours) is information that might need to be passed to other functions (such as selecting direction based on neighbours).

Without knowing any more details about these functions, we can write the top level process in the Update function, and create the definitions of the various functions (with details of parameters, and comments explaining what they do). This structure is shown in Algorithm 19. This program does not do anything until the individual functions are written, but we at least have a readable idea of what the overall process is, and what the individual steps are.

Algorithm 19 Top level functions for boid simulation.

```
1 void Update () {
2     listOfNeighbours = findNeighbours ();
3     targetDirection = calculateDirection (
4         listOfNeighbours);
5     moveDirection = checkBounds (targetDirection);
6     moveOneStep (moveDirection);
7 }
8 // Return the list of neighbours within a particular
// distance of the current object.
9 NeighbourList findNeighbours ()
10 {
11 }
12
13 // Use the boids direction calculation based on the
// given neighbour's positions.
14 Vector3 calculateDirection (NeighbourList neighbours)
15 {
16 }
17
18 // Ensure that a movement in the given direction does
// not take the object beyonds the bounds of the
// movement area.
19 Vector3 checkBounds (Vector3 direction)
20 {
21 }
22
23 // Take a step in the given direction at a particular
// speed.
24 void moveOneStep (Vector3 direction)
25 {
26 }
```

4. This representation also allows us to fill in some details that we might have glossed over. Some functions need extra parameters to avoid having to enter magic constants into them. These include speed of movement (for moveOneStep), the boundary positions (for checkBounds), and the radius to check for neighbours (findNeighbours). We add these as parameters, and set the values when calling the functions (in Update). Some of these values can then be exposed to the Unity software editor as public variables.

We can also take this opportunity to work out what type to use of a list of neighbouring objects. Unity software supports an OverlapSphere function that returns all objects within a set distance. This returns a list of Collider objects, so we'll use this. The script can now be updated to a runnable (but non-functional) version as shown in Algorithm 20. This version should allow the virtual reality application to run. It does not do anything yet, but we now have the opportunity to develop each function in turn, and test it individually.

5. We'll start with the findNeighbours function. As mentioned, we'll use the Unity software OverlapSphere function to do this. Since this is the only step required, we won't break down the actions of findNeighbours into any smaller functions. The code below shows an initial attempt:

```

1  Collider [] hitColliders = Physics.OverlapSphere (
2      this.transform.position, radius);
3  print ("Colliders: " + hitColliders + ", " +
       hitColliders.Length);
4  return hitColliders;

```

Note we include a print function (section 4.2 on page 145) so that we can check to see what the program is actually doing. Run the program and see what the results look like.

My version finds 7 objects when I've only added 6 boids prefabs to the scene. I suspect it is finding the

Algorithm 20 Runnable skeleton of Boids simulation.

```

1 // the radius of a neighbourhood.
2 public float neighbourRadius = 0.5f;
3
4 // size of boundary area.
5 public float boardSize = 5.0f;
6
7 // speed of boid movement
8 public float speed = 1.0f;
9
10 // Update is called once per frame
11 void Update () {
12     Collider [] listOfNeighbours = findNeighbours (
13         neighbourRadius);
14     Vector3 targetDirection = calculateDirection (
15         listOfNeighbours, 0.3f, 0.1f, 0.2f);
16     Vector3 moveDirection = checkBounds (targetDirection
17         , boardSize);
18     moveOneStep (moveDirection, speed);
19 }
20
21 // Return the list of neighbours within a particular
22 // distance of the current object.
23 Collider [] findNeighbours (float radius)
24 {
25     return null;
26 }
27
28 // Use the boids direction calculation based on the
29 // given neighbour's positions.
30 Vector3 calculateDirection (Collider [] neighbours,
31     float separation, float convergenceFactor, float
32     alignmentFactor)
33 {
34     return new Vector3 (0,0,0);
35 }
36
37 // Ensure that a movement in the given direction does
38 // not take the object beyonds the bounds of the
39 // movement area.
40 Vector3 checkBounds (Vector3 direction, float limit)
41 {
42     return new Vector3 (0,0,0);
43 }
44
45 // Take a step in the given direction at a particular
46 // speed.
47 void moveOneStep (Vector3 direction, float speed)
48 {
49 }
```

sky layer. I'll check this out by removing the sky layer temporarily and see if the result changes. Having confirmed that theory, we need to find boids only. The OverlapSphere function supports selecting only objects on a particular layer. Create a new layer (under the Inspector pane) and set the boid prefab to use this. My version uses layer 9. We can use the version of OverlapSphere that takes a layer as parameter. This update seems to work as desired, so we update the comments for the findNeighbours function to indicate the layer restriction that it uses.

6. Some other functions are also easy 1-liners. MoveOnStep is achieved using a simple kinematics simulation (section 7.2 on page 254). We'll modify the calculateDirection and checkBounds functions to generate a non-zero vector and to pass this through (respectively) so we can test this. With the changes below, the individual boids end up moving in a fixed direction.

```

1 Vector3 calculateDirection (Collider [] neighbours,
                            float separation, float convergenceFactor,
                            float alignmentFactor)
2 { return new Vector3 (1,0,0); }
3
4 Vector3 checkBounds (Vector3 direction, float limit
                      )
5 { return direction; }
6
7 void moveOneStep (Vector3 direction, float speed)
8 {
9     Vector3 acceleration = direction;
10    velocity = velocity + acceleration * Time.
11        deltaTime;
12    // some viscosity.
13    velocity = (1.0f - (0.2f * Time.deltaTime)) *
14        velocity;
15    if (velocity.magnitude > speed)
16    {
17        velocity = speed * velocity.normalized;
18    }
19    this.transform.position = this.transform.position
20        + velocity * Time.deltaTime;
21    this.transform.forward = velocity;
22 }
```

-
7. The calculateDirection function is a more complex

task and is broken down into further steps that represent additional functions that are called:

- (a) Get the average position of the neighbours
- (b) Get the average direction of the neighbours
- (c) Calculate the direction to the average position of the neighbours if we're more than separation distance, or directly away from this if we're closer.
- (d) Combine direction to average position, and average direction, in the ratio of convergenceFactor to alignmentFactor.

The function would look like:

```

1  Vector3 calculateDirection (Collider [] neighbours,
2      float separation, float convergenceFactor,
3      float alignmentFactor)
4  {
5      Vector3 averagePosition = findAveragePosition (
6          neighbours);
7      Vector3 averageDirection = findAverageDirection (
8          neighbours);
9      Vector3 directionToAveragePosition =
10         averagePosition - this.transform.position;
11      if (directionToAveragePosition.magnitude <
12          separation)
13      {
14          // we're too close - change direction to away
15          // from the center.
16          directionToAveragePosition = -
17              directionToAveragePosition;
18      }
19      directionToAveragePosition =
20          directionToAveragePosition.normalized;
21      Vector3 direction = directionToAveragePosition *
22          convergenceFactor + averageDirection *
23          alignmentFactor;
24
25      print ("Direction: " + direction + " -- " +
26          averagePosition + "    " + averageDirection +
27          "    " + directionToAveragePosition);
28      return direction;
29 }
```

Average position and direction are calculated using functions that iterate over the neighbours and calculate an average. This code also includes a print statement

so that we can monitor the values calculated, and check that they seem plausible.

8. The function to checkBounds involves testing to see if the motion has moved beyond the limits of the background sky plane. This function involves the steps shown below (including some fixes as a result of testing: a bit of random perturbation to resolve cases where all objects end up on top of one another, and another to stop rounding effects pushing objects into the plane).

```

1  Vector3 checkBounds (Vector3 direction, float limit
)
2  {
3      // very small perturbation to break any ties.
4      direction = direction + new Vector3 (Random.Range
5          (-0.01f,0.01f), Random.Range (-0.01f,0.01f),
6          0);
7
8      // remove any z-component.
9      direction.z = 0;
10
11     if (this.transform.position.x < -limit)
12     {
13         direction.x = Mathf.Abs (direction.x);
14         velocity.x = Mathf.Abs (velocity.x);
15     }
16     if (this.transform.position.x > limit)
17     {
18         direction.x = -Mathf.Abs (direction.x);
19         velocity.x = -Mathf.Abs (velocity.x);
20     }
21     if (this.transform.position.y < -limit)
22     {
23         direction.y = Mathf.Abs (direction.y);
24         velocity.y = Mathf.Abs (velocity.y);
25     }
26     if (this.transform.position.y > limit)
27     {
28         direction.y = -Mathf.Abs (direction.y);
29         velocity.y = -Mathf.Abs (velocity.y);
30     }
31
32     return direction;
33 }
```

This version of the function still has a number of repeated elements that might be achieved more efficiently with multiple calls to a single parameterized function. This change is shown below. A single func-

tion that takes in an axis parameter indicating the direction where a limit needs to be enforced. The vector dot product determines how far along this axis the object actually is, and if it exceeds that limit, sets the direction and velocity vectors to point in the opposite direction with respect to that axis by adding twice the reverse direction (once to cancel, once to force the opposite direction).

```

1  void enforceLimit (ref Vector3 direction, ref
                      Vector3 velocity, float limit, Vector3 axis)
2  {
3      if (Vector3.Dot (this.transform.position, axis) >
               limit)
4      {
5          direction += -2.0f * Mathf.Abs (Vector3.Dot (
                           direction, axis)) * axis;
6          velocity += -2.0f * Mathf.Abs (Vector3.Dot (
                           velocity, axis)) * axis;
7      }
8  }
9
10 // Ensure that a movement in the given direction
11 // does not take the object beyonds the bounds of
12 // the
13 // movement area.
14 Vector3 checkBounds (Vector3 direction, float limit
15 )
16 {
17     // very small perturbation to break any ties.
18     direction = direction + new Vector3 (Random.Range
19         (-0.01f,0.01f), Random.Range (-0.01f,0.01f),
20         0);
21     // remove any z-component.
22     direction.z = 0;
23
24     enforceLimit (ref direction, ref velocity, limit,
25                     new Vector3 (1,0,0));
26     enforceLimit (ref direction, ref velocity, limit,
27                     new Vector3 (-1,0,0));
28     enforceLimit (ref direction, ref velocity, limit,
29                     new Vector3 (0,1,0));
30     enforceLimit (ref direction, ref velocity, limit,
31                     new Vector3 (0,-1,0));
32
33     return direction;
34 }
```

9. The complete set of functions involved are shown in Algorithm 21 on page 116, 22 on page 117, 23 on

page 118, 24 on page 119 and 25 on page 120. The use of abstraction has allowed us to build quite a complex system by decomposing the problem into individual functions with clearly defined roles. Each function can then be tested individually. An example of a flock formed after a period of time is shown in Figure 3.5.2.

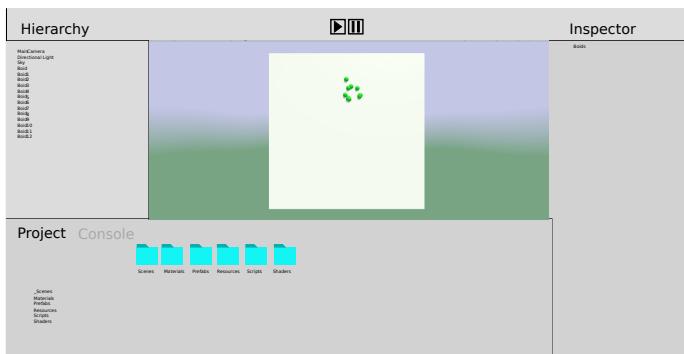


Figure 3.5.2: A view of a flocking set of objects, maintaining a coherent but dynamic behaviour.

3.6 Collections

3.6.1 Description

The variables encountered so far are good for storing a single value. When we assign a new value to them, the old value is discarded and overwritten with the new. There are occasions when we need to keep track of a collection of related items. A popular example is keeping track of the position of treasure items that a user needs to collect while exploring a virtual world.

There are variable types that can store collections of objects. The simplest of these is the array which is supported by most programming languages. Other collection types include lists, queues, vectors and sets. The differences between these relate to the ease with which elements can be added and removed, and the order in which elements are stored and retrieved. Most collections do support common conventions for declaring the col-

Algorithm 21 Complete Boid solution (part 1).

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class BoidMovement : MonoBehaviour {
6
7      // the radius of a neighbourhood.
8      public float neighbourRadius = 0.5f;
9
10     // size of boundary area.
11     public float boardSize = 5.0f;
12
13     // an upper bound on the speed of boid movement
14     public float speed = 1.0f;
15
16     // internal variables needed for kinematic
17     // simulation.
17     private Vector3 velocity;
18
19     void Start () {
20         velocity = new Vector3 (0,0,0);
21     }
22
23     // Update is called once per frame
24     void Update () {
25         Collider [] listOfNeighbours = findNeighbours (
26             neighbourRadius);
26         Vector3 targetDirection = calculateDirection (
27             listOfNeighbours, 0.45f, 0.4f, 0.2f);
27         Vector3 moveDirection = checkBounds (
28             targetDirection, boardSize);
28         moveOneStep (moveDirection, speed);
29     }
30
31     // Return the list of neighbours within a particular
32     // distance of the current object. Only objects on
32     // layer 9
33     // are selected.
34     Collider [] findNeighbours (float radius)
35     {
36         Collider [] hitColliders = Physics.OverlapSphere (
36             this.transform.position, radius, 1 << 9);
37         return hitColliders;
38     }
```

Algorithm 22 Complete Boid solution (part 2).

```
1 // Use the boids direction calculation based on the
2 // given neighbour's positions.
3 Vector3 calculateDirection (Collider [] neighbours,
    float separation, float convergenceFactor, float
    alignmentFactor)
4 {
5     Vector3 averagePosition = findAveragePosition (
        neighbours);
6     Vector3 averageDirection = findAverageDirection (
        neighbours);
7     Vector3 directionToAveragePosition = averagePosition
        - this.transform.position;
8     if (directionToAveragePosition.magnitude <
        separation)
9     {
10         // we're too close - change direction to away from
            the center.
11         directionToAveragePosition = -
            directionToAveragePosition;
12     }
13     directionToAveragePosition =
            directionToAveragePosition.normalized;
14     Vector3 direction = directionToAveragePosition *
            convergenceFactor + averageDirection *
            alignmentFactor;
15
16     return direction;
17 }
```

Algorithm 23 Complete Boid solution (part 3).

```

1 // Find the average position of the objects in the
   list of neighbours.
2 Vector3 findAveragePosition (Collider [] neighbours)
3 {
4     Vector3 averagePosition = new Vector3 (0,0,0);
5     // add each element.
6     for (int i = 0; i < neighbours.Length; i++)
7     {
8         averagePosition = averagePosition + neighbours[i].
           gameObject.transform.position;
9     }
10    // divide by the number of elements.
11    averagePosition = (1.0f / neighbours.Length) *
           averagePosition;
12    return averagePosition;
13 }
14
15 // Find the average direction of the objects in the
   list of neighbours.
16 // Returns a normalized vector, unless the average
   direction is a zero vector.
17 Vector3 findAverageDirection (Collider [] neighbours)
18 {
19     Vector3 averageDirection = new Vector3 (0,0,0);
20     // add each element.
21     for (int i = 0; i < neighbours.Length; i++)
22     {
23         averageDirection = averageDirection + neighbours[i].
           gameObject.transform.forward;
24     }
25     // divide by the number of elements.
26     averageDirection = (1.0f / neighbours.Length) *
           averageDirection;
27     // normalize
28     if (averageDirection.magnitude != 0.0f)
29     {
30         averageDirection = averageDirection.normalized;
31     }
32     return averageDirection;
33 }
```

Algorithm 24 Complete Boid solution (part 4).

```
1 void enforceLimit (ref Vector3 direction, ref Vector3
2   velocity, float limit, Vector3 axis)
3 {
4   if (Vector3.Dot (this.transform.position, axis) >
5     limit)
6   {
7     direction += -2.0f * Mathf.Abs (Vector3.Dot (
8       direction, axis)) * axis;
9     velocity += -2.0f * Mathf.Abs (Vector3.Dot (
10      velocity, axis)) * axis;
11 }
12
13 // Ensure that a movement in the given direction
14 // does not take the object beyonds the bounds of the
15 // movement area.
16 Vector3 checkBounds (Vector3 direction, float limit)
17 {
18   // very small perturbation to break any ties.
19   direction = direction + new Vector3 (Random.Range
20     (-0.01f,0.01f), Random.Range (-0.01f,0.01f), 0)
21   ;
22   // remove any z-component.
23   direction.z = 0;
24
25   enforceLimit (ref direction, ref velocity, limit,
26     new Vector3 (1,0,0));
27   enforceLimit (ref direction, ref velocity, limit,
28     new Vector3 (-1,0,0));
29   enforceLimit (ref direction, ref velocity, limit,
30     new Vector3 (0,1,0));
31   enforceLimit (ref direction, ref velocity, limit,
32     new Vector3 (0,-1,0));
33
34   return direction;
35 }
```

Algorithm 25 Complete Boid solution (part 5).

```

1   // Take a step in the given direction at a
2   // particular
3   void moveOneStep (Vector3 direction, float speed)
4   {
5       Vector3 acceleration = direction;
6       velocity = velocity + acceleration * Time.
7           deltaTime;
8       // some viscosity.
9       velocity = (1.0f - (0.2f * Time.deltaTime)) *
10      velocity;
11      if (velocity.magnitude > speed)
12      {
13          velocity = speed * velocity.normalized;
14      }
15      this.transform.position = this.transform.position
16      + velocity * Time.deltaTime;
17      this.transform.forward = velocity;
18  }

```

lection and specifying initial size (if required), accessing particular elements, and adding and removing elements.

Once a collection has been created we need to manipulate it. The types of collection manipulation patterns that are commonly used include:

- Updating or accessing every element in the collection.
- Checking to see if a particular element is in the collection.
- Adding an element to the collection.
- Removing an element from the collection

3.6.2 Pattern

A collection would be defined by declaring a variable of one of the collection types. The pattern would be:

```

CollectionType collectionName =
    new CollectionType [CollectionSize]

```

Exact syntax differs depending on language and nature of the collection. Arrays generally require the size of the array to be known in advance.

Individual elements in the collection are addressed (either to retrieve their value, or to assign a value to them) using a notation such as:

```
collectionName[i]
```

This refers to the i^{th} element of the collection. Collections are usually numbered from 0 upto CollectionSize - 1. Thus the first element is collectionName[0].

The collection type is any of the supported types in the language. This includes primitive types such as integer and floating point values, engine defined types such as vectors and quaternions, but also user defined types and classes, as well also as other collection types.

An iteration pattern is used to operate on each element of the collection. Some languages support particular iterator variables to traverse the collection.

```
for (i = 0; i < CollectionSize; i++)
// or an alternative if the language
// supports iterators.
// foreach element in collectionName
{
    // retrieve ith element, not required
    // if using iterators.
    element = collectionName[i]
    // modify the ith element
    collectionName[i] = ... // new value
}
```

Checking to see if an element exists in a collection can make use of particular functions defined for the collection (such as Find) if they exist, or can be achieved by checking each element of the list manually while iterating over the collection. This can take some time, particularly if the list is long, so should be used sparingly.

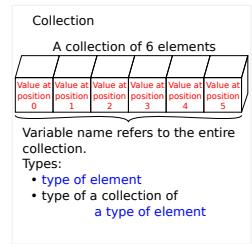


Figure 3.6.1: A collection is regarded as a variable capable of holding multiple values, each identified by their index in the container.

Rather find the element once and store the result, than search for the same element on every frame update.

```
boolean elementExists = false
CollectionType foundElement = null
foreach element in collectionName
{
    if element satisfies requirements
    {
        foundElement = element
        elementExists = true
        break
    }
}
// Code below checks that the element was found
// before doing anything with it
if (elementExists)
{
    // do something with foundElement
}
else
{
    // no matching element was found
}
```

This process starts at the beginning of the collection and checks each element in turn to see if it is the one we want. We typically use some form of search criterion (e.g. element has a particular name, or a particular property has a desired value) rather than checking for equality to another element or copy of the target element. For most objects, testing for equality between an object and a copy of that object returns false, since they are regarded as two separate and different objects. Checking to see if the values of their properties match is often what is intended.

At the end of the loop, the Boolean variable *elementExists* is true if a suitable element has been found, and the reference to that element is in *foundElement*. The break statement in the loop allows the iteration to terminate once a matching element has been found, so saving time needlessly checking the rest of the list.

Adding an element to a collection usually uses an Add function defined for the collection. Adding elements to arrays which have a fixed and predefined size requires keeping track of the number of elements in the array. So initially:

```
int numberOfElements = 0
```

When adding a new element:

```
if (numberOfElements < CollectionSize)
{
    collectionName[numberOfElements] = element
    numberOfElements = numberOfElements + 1
}
```

It is a good idea to check if the array is full before adding new elements. Bad things can happen when adding elements at array positions outside the allowed range.

Removing an element from a collection can make use of a Remove function, provided a reference to the element is available. Using a loop to find and remove elements from a collection is hazardous because removing an element from a collection changes its length, which affects the loop counter, which means the loop may step over some elements. It is better to first find the elements to be removed, and then delete them in a separate step (i.e. not in the loop that searches for the elements).

3.6.3 Example

Example 17. Applying the pattern to Unity software

1. A popular activity in many virtual reality applications is based around a treasure hunt theme where users need to wander around picking up objects of a particular type (such as coins) from the scene. This example uses collections to distribute and track the collection of a set of coins.

This facility is added to a project that already has some object that can collect the coins, such as a controllable avatar character (see the example in section 2.6 on page 47). We'll be using the collision detection facility to detect when coins are collected, so this character needs a collider component.

Create a coin object. This needs a Rigidbody component added. The coin needs the "Is Trigger" box checked in the collider component and the "Use Gravity" box unchecked in the Rigidbody component. Add a C# component called CoinActivity to the coin. Convert the coin to a prefab.

2. The list of coins is managed using a manager pattern (section 2.9 on page 69). Create an empty object to be the coin manager. The script for this is given in Algorithm 26 on the facing page. This script defines a collection of coins using a List container type. The type of the objects stored in this list needs to be specified, within the <> brackets. This container needs to be initialized in the Start method. During the loop that instantiates each of the coins, the game object returned by the Instantiate function is added to the list, using the list's Add function.

The *coinList* variable keeps track of the coins remaining in the scene. Before any coin is removed from the scene (in the registerCoinCollect function) it must first be removed from this list, before being deleted in the scene graph. This ensures that the list tracks the changes in the scene. When the list becomes empty then we know that no coins are left in the scene. The script demonstrates a simple check for this scenario,

Algorithm 26 Collection management involving instantiating various objects while keeping track of these in a list.

```
1 ...
2 public class CoinTracker : MonoBehaviour {
3     // Number of coins to generate.
4     public int NumberOfCoins = 10;
5     // Size of area to scatter them.
6     public float radius = 10.0f;
7     // Template for a coin.
8     public GameObject coinPrefab;
9     // Internal list to keep track of coins.
10    private List<GameObject> coinList;
11
12    // Use this for initialization
13    void Start () {
14        // Create the coin list.
15        coinList = new List<GameObject> ();
16
17        // Instantiate some coins when the application
18        // starts.
19        for (int i = 0; i < NumberOfCoins; i++)
20        {
21            GameObject coin = Instantiate (coinPrefab,
22                new Vector3 (Random.Range (-radius, radius),
23                    0.6f, Random.Range (-radius, radius)),
24                Quaternion.AngleAxis (90, new Vector3 (1,0,0))
25            );
26            coin.GetComponent <CoinActivity> ().manager =
27                this;
28            coinList.Add (coin);
29        }
30    }
31
32    public void registerCoinCollect (GameObject coin)
33    {
34        coinList.Remove (coin);
35        Destroy (coin);
36
37        if (coinList.Count == 0)
38    }
```

restraining its jubilation to just printing a message to the console log.

3. Collecting the coins occurs when an object collides with a coin. Collision detection events are triggered in the individual coins, rather than in the coin manager where we need to update the collection. To ensure that the coin informs the manager of any collisions, we add a public manager property to the CoinActivity component attached to the coins. When the collision occurs, the coin (through CoinActivity) calls the public registerCoinCollect function in the coin manager.

The manager variable is defined in CoinActivity with:

```
1  public CoinTracker manager;
```

CoinActivity also includes an event callback to invoke a function on the manager when a collision occurs. OnTriggerEnter is an event that is generated when the collision first starts, provided that the “Is Trigger” checkbox on the collider is ticked.

```
1  void OnTriggerEnter(Collider other) {
2      if (manager != null)
3      {
4          manager.registerCoinCollect (this.gameObject);
5      }
6 }
```

3.7 Fading with distance: Visibility, Scope and Lifetime

3.7.1 Description

It is possible to declare variables with the same names as variables that have been used elsewhere in the program, but in different blocks. While the version in the closest encompassing block is the one that is used in such cases, it is then worth being aware of the rules around access to variables within the program.

Visibility and scope refer to the parts of the program that are able to access a particular variable. Usually

scope and visibility are the same thing, except in rare circumstances such as that mentioned previously where a variable name is reused for a different variable. The visibility of a variable is limited to the block in which it is declared. This includes any blocks enclosed inside this block. Parameters for a function are visible only within the block containing the function code.

Attributes of a class are visible within the block of the class itself which includes all the methods of the class. Additional modifiers are applied to extend the scope of attributes beyond the class (public), or enforce the restriction that attributes are only visible within the class (private). Function names also conform to these scope rules.

Variables defined outside a block are considered to have global scope. Global variables are visible anywhere in the application code, although some languages restrict these variables to being global only for the remainder of the file in which they are declared. Special modifiers then have to be used to achieve globally global variables.

The lifetime of the variable is the period for which it exists. Variables may still retain their values even when they are not in scope such as the variables with duplicate names scenario mentioned previously. While the new version may be in scope, both versions are still alive. When the new variable comes to the end of its scope, and lifetime, the old variable returns to scope with its original value intact as it was alive the entire time.

A variable's lifetime usually comes to an end when execution reaches the end of the block in which it was declared. Special modifiers exist in some languages (static) to preserve the variable (extend its lifetime) until the next time that particular block of code is run.

The lifetime of global variables is the entire program's execution time.

Objects that are created with the new operator have a lifetime that extends from their creation time until they

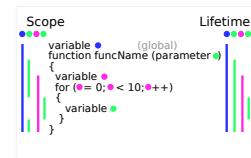


Figure 3.7.1: Scope and lifetime may appear to be similar but scope identifies if a variable can be accessed (or which version is accessed) at a particular point in the code, and lifetime indicates whether the variable retains its value at that point.

are explicitly freed up, or until the last reference to them is removed in languages that perform garbage collection. The variables that refer to objects are not objects themselves but just references and their lifetime is limited to their local block as is the case for other variables.

3.7.2 *Pattern*

Various examples of scope are illustrated in the pattern below:

```
int globalVariable
class MyClass
{
    private:
        int classLimitedVariable
        function classLimitedFunction ()
        {
        }

    public:
        int globalVariableInClass
        function globalFunctionInClass ()
        {
        }
        function generalFunction (int parameter)
        {
            int localToGeneralFunction
            classInstance = new MyClass
        }
}

MyClass classInstance // file or global scope
```

The global attribute and method belonging to the class are in scope everywhere but do need to be accessed using the object reference for that specific instance. For example:

```
classInstance.globalVariableInClass
classInstance.globalFunctionInClass ()
```

The scope of parameter is limited to the block defined for generalFunction.

The lifetime of each of the variables is:

- globalVariable: for the duration of the program
- classLimitedVariable, globalVariableInClass: from the time the object is instantiated until it is destroyed. In the particular case of reference variable classInstance the reference is global so is created when the program starts. However the object it refers to is only created and assigned to the reference when generalFunction is run, which is when the lifetime of the class attributes in that instance starts. The reference then appears to persist unchanged until the program stops running, at which point the lifetime of classInstance, the object referred to by classInstance, and the class attributes come to an end.
- parameter, localToGeneralFunction: the lifetime starts when generalFunction is called and ends when generalFunction finishes.

3.7.3 Example

Example 18. Applying the pattern to Unity software

1. In this example we'll explore a common scenario in virtual reality applications: communication between objects. This is quite a sensitive topic because unconstrained communication between any pair of objects results in incredibly complex code that is almost impossible to maintain and extend. We should be focusing on object-oriented design principles and designing each class of object to provide a minimal public interface. Ideally we should reduce the need for other objects to have specific knowledge of the nature of, or internal workings, of others.

Modifiers to visibility are a good way to achieve a well defined interface. Variables and methods should be private by default, and only made public if:

- They are intended to be part of the public interface, and used by external objects. Ideally use functions for the public interface, rather than exposing individual variables since the code within a function is more easily updated than a dependency on accessing a variable of a particular type and name. In the interests of abstraction, expose functions that perform the most common operations required so that external objects don't have to each redefine the same processes.
- In the case of Unity software, any variables that need to be defined in the Unity software editor. Ideally these are just initial values to be used at startup and are not also intended to be manipulated while the simulation is running.

The example shown here is a rather contrived simulation. A car object must ask a bridge object to lower/raise itself as the car approaches/leaves.

2. The scene configuration is at the developer's discretion. However there are two key objects: a car, and a bridge manager. Each needs its own C# component, named Car and Bridge respectively.

The car is responsible for driving itself. It decides at appropriate points on its trip that it wants to open and close the bridge. Since it is requesting functionality from the bridge it needs to know about the bridge. In this case, we provide a public variable in the car, that we assign the bridge component to in the Unity software Editor. There are other ways of finding the bridge, ranging from searching the scene graph for an object with the appropriate name or label, to adding a collider to the entrance and exit of the bridge and finding the bridge component from that.

We keep the car as simple as possible for this example as shown in the code in Algorithm 27. Note that this code is not the most elegant or robust. It only works

for a car going in one direction, and continually tries to open or close the bridge on every update once one of the conditions are satisfied. The best that could be said of it is that it checks if a value has actually been assigned to the bridge variable before using it.

3. The bridge must provide the public interface expected from it; specifically an open function, and a close function. These need to be public since they are being accessed from the car object, which is external to the bridge component of the bridge manager object. In an effort to atone for the shoddy code in the car we try to make our public functions robust since we have limited control over how they are used. We keep track of the bridge's state, and only update the geometry if the state has actually changed. This state is specific to the bridge so we keep it private.

The bridge needs two public variables to access the sections of roadway that it needs to control. The open and close functions just change the rotation of these and updates the internal state variable, if required. The script for this is shown in Algorithm 28. Figure 3.7.2 shows several steps in the sequence of actions.

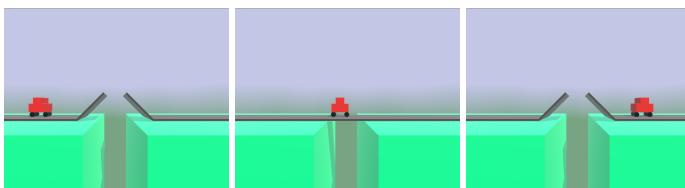


Figure 3.7.2: The interface exposed by the bridge responds to instructions from the car object to first close and then reopen.

3.8 Recursion or I've seen this somewhere before

3.8.1 Description

The solution to some problems can involve solving smaller versions of the same problem. For example, given the remarkable self-similarity in structures such as trees, the

Algorithm 27 The car invokes functions on the bridge object to control the bridge.

```
1  public class Car : MonoBehaviour {
2
3      // the reference to the bridge component on the
4      // bridge manager so we can use its public
5      // functions.
6
7      public Bridge bridge;
8
9
10     // Update is called once per frame
11     void Update () {
12
13         // we assume the car starts on the left (negative
14         // x position) and moves right by increasing x.
15         this.transform.position += new Vector3 (0.01f, 0,
16             0);
17
18         if (bridge != null)
19         {
20             // when the car has crossed the bridge, open it
21             // (up)
22             if (this.transform.position.x > 3.0f)
23             {
24                 bridge.open ();
25             }
26             // when the car gets past the start of the
27             // bridge, close it (down)
28             else if (this.transform.position.x > -3.0f)
29             {
30                 bridge.close ();
31             }
32         }
33     }
```

Algorithm 28 The public interface of the bridge consists of two functions: open and close, which manipulate the internal private state variables of this object.

```
1  public class Bridge : MonoBehaviour {
2
3      // variables provided by the Unity software editor
4      public GameObject bridgeSection1;
5      public GameObject bridgeSection2;
6
7      // private internal state of the bridge.
8      private bool bridgeOpen;
9
10     // Use this for initialization
11     void Start () {
12         // start with the bridge open.
13         bridgeOpen = false;
14         open ();
15     }
16
17     public void open ()
18     {
19         // only open the bridge if it is closed.
20         if (!bridgeOpen)
21         {
22             // rotate the bridge sections.
23             bridgeSection1.transform.rotation = Quaternion.
24                 AngleAxis (45.0f, new Vector3 (0,0,1));
25             bridgeSection2.transform.rotation = Quaternion.
26                 AngleAxis (-45.0f, new Vector3 (0,0,1));
27             // update the internal state since the bridge is
28             // now open.
29             bridgeOpen = true;
30         }
31     }
32
33     public void close ()
34     {
35         if (bridgeOpen)
36         {
37             bridgeSection1.transform.rotation = Quaternion.
38                 AngleAxis (0.0f, new Vector3 (0,0,1));
39             bridgeSection2.transform.rotation = Quaternion.
40                 AngleAxis (0.0f, new Vector3 (0,0,1));
41             bridgeOpen = false;
42         }
43     }
44 }
```

² <http://algorithmicbotany.org/papers/#abop>

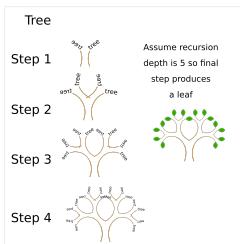


Figure 3.8.1: A tree is one structure that contains self-similarity encouraging recursive strategies for creating and manipulating such structures.

problem of drawing a tree could be solved by drawing a branch, and then drawing two small trees at the end of the branch [Prusinkiewicz and Lindenmayer, 1996]².

Unlike the sorcerer's apprentice we actually need a place to stop, otherwise the process will continue until we destroy the universe by penetrating the quantum foam with picoscopic trees. Once we get down to a small enough tree/branch, we can just draw one leaf instead.

3.8.2 Pattern

The pattern for recursion involves defining a function that represents a solution to the problem. This function:

- contains a base case: this is the stopping condition. In the base case we perform non-recursive actions; actions that do not lead to invoking the function itself.
- contains the recursive path: this might involve doing some small and easily achievable work (drawing a branch) followed by a call to this same function to solve a smaller version of the same problem.

We would typically pass in a parameter to this function to indicate the scale we are working at. The base case checks if this parameter has reached a threshold (too small). Otherwise the recursive path reduces the parameter slightly and passes this new value to the next invocation of the function.

The recursive pattern is:

```
function recursiveFunction (scaleParameter,
    otherParameters)
{
    if (scaleParameter < threshold)
    {
        // base case.
        do something simple
    }
    else
```

```

{
    // recursive path
    do something else simple
    recursiveFunction (scaleParameter - 1,
        otherParameterValues1)
    // Can recurse multiple times if the
    // problem requires e.g. if we have
    // to draw two smaller trees.
    recursiveFunction (scaleParameter - 1,
        otherParameterValues2)
}
}

```

Try to trace the lifetime and scope of the variables used to see if you truly understand all of these concepts.

3.8.3 Example

Example 19. Applying the pattern to Unity software

1. The use of a recursive function to generate a tree is shown in Algorithm 29 and 30.

In terms of the recursive pattern:

- The base case occurs when the variable *levels* reaches 0. In that case, the tree is just a leaf.
- The recursive path (when *levels* > 0) involves creating a branch of the current trunk length, and in the current direction. A loop is then used to create each of the sub-branches. The direction of the subbranch is determined by modifying the current direction with rotations about:
 - the up direction: so all *n* branches are spaced at equal angles of $\frac{360^\circ}{n}$ around the parent branch.
 - about the side (right) direction, so that each branch bends down relative to the parent.

Each subbranch is then just a tree with one less level.

Algorithm 29 A recursive function to procedurally generate a tree like structure (part 1).

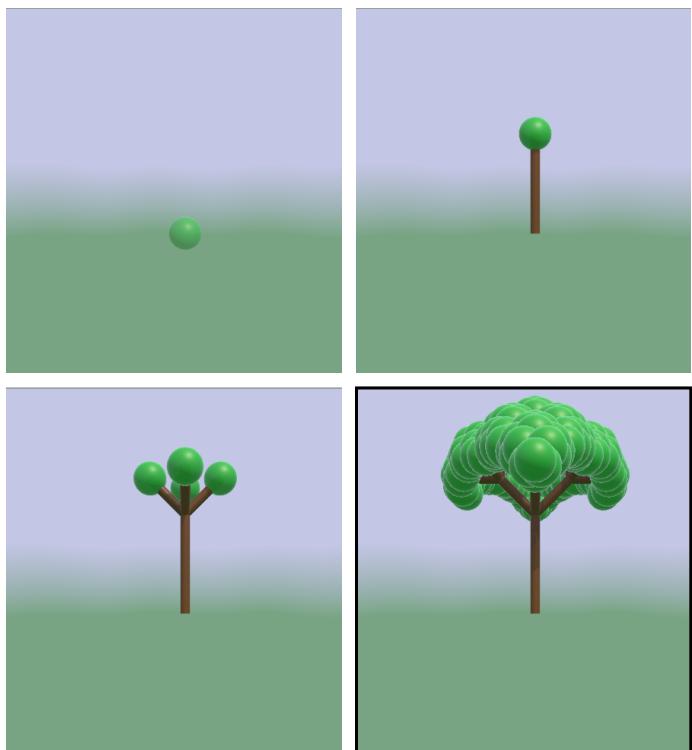
```
1  ...
2  public class TreeGenerator : MonoBehaviour {
3
4      [Tooltip ("For example, use a unit scale sphere at
5          origin.")]
6      public GameObject leafPrefab;
7      [Tooltip ("For example, use a unit cylinder from
8          origin to y=1.")]
9      public GameObject branchPrefab;
10
11     [Tooltip ("The number of levels of recursion (number
12         of generations of branches before we get to a
13         leaf")]
14     public int levels = 5;
15     [Tooltip ("How many banches branch out of a branch."
16         )]
17     public int branchFactor = 4;
18     [Tooltip ("The angle between a branch and the next (
19         sub)branch.")]
20     public float branchAngle = 45.0f;
21     [Tooltip ("The initial scale (length) of the trunk."]
22     public float trunkLength = 3.0f;
23     [Tooltip ("The length of a subbranch relative to the
24         length of its parent.")]
25     public float trunkDecayFactor = 0.5f;
26
27     void Start () {
28         generateTree (levels, transform.position,
29                         transform.rotation, trunkLength);
30     }
31     // continued ...
```

Algorithm 30 A recursive function to procedurally generate a tree like structure (part 2).

```
1  // continuation ...
2  void generateTree (int levels, Vector3 position,
                     Quaternion direction, float trunkLength)
3  {
4      if (levels == 0)
5      {
6          GameObject leaves = Instantiate (leafPrefab,
                                         position, direction);
7      }
8      else
9      {
10         GameObject branch = Instantiate (branchPrefab,
11                                         position, direction);
12         branch.transform.localScale = new Vector3 (1,
13                                         trunkLength, 1);
14         Vector3 endOfBranchPosition = branch.transform.
15             position + trunkLength * branch.transform.
16             up;
17         for (int i = 0; i < branchFactor; i++)
18         {
19             Quaternion newBranchDirection = direction *
20                 Quaternion.AngleAxis (i * 360 /
21                                         branchFactor, Vector3.up);
22             newBranchDirection = newBranchDirection *
23                 Quaternion.AngleAxis (branchAngle,
24                                         Vector3.right);
25             generateTree (levels - 1, endOfBranchPosition,
26                           newBranchDirection, trunkLength *
27                           trunkDecayFactor);
28         }
29     }
30 }
```

2. With respect to the coding style and procedural reasoning embodied in this example you may wish to note:
 - The use of the tooltip attribute to replace comments for the public variables that are accessible to the Unity software editor. This serves the dual purpose of documenting the variable to the developer, but also making this documentation available to the person using the component. Since the code is self-documented in only one place there is a lower chance of the documentation getting out of date than if a separate user manual had to be written.
 - The process is extensively controlled by parameters. This allows the appearance of any particular tree to be customized using these variables, ensuring that the same code can be reused for creating a forest of varied individuals.
 - This style of procedurally generating trees is a simplified version of the L-systems [Prusinkiewicz and Lindenmayer, 1996] approach historically used for creating plants. L-systems allow the particular sets of rules (grammatical productions) to be customized according to the characteristics of each species of plant. This example is effectively an L-system with 2 rules: either a branch becomes a leaf, or a branch becomes a branch followed by n branches. Modern plant generation processes also tend to include environmental effects since the growth of a tree is affected by light and other resources. This resource access in turn is controlled by the light blocking obstacles in the scene, which can include other trees.
3. Examples of trees produced for different initial values of level are shown in Figure 3.8.2.

Figure 3.8.2: Trees generated with initial levels of recursion of 0, 1, 2 and 5.



4

What Could Go Wrong?

4.1 Debugging

Regardless of what you may see your teacher doing, programs do not flow fully formed from fingertips to keyboard, and then operate perfectly flawlessly first time around. There are several intermediate stages which may often be obscured by the need to complete a class on time, or the desire not to appear to be a bumbling idiot in front of an audience.

The stages in developing virtual reality applications include:

Design: this is not the visual design (although that has its place as well), but rather the functional design. The key components of the VR application are sketched out here, along with identifying what the function of each component must be. At this stage it is well worth considering what information is required by each component, what information needs to be communicated between components and what transformations of this information occur within each component. Recall that behaviour in a virtual reality application is equivalent to modifying information (in the scene graph) and so being able to match the functionality of the application with the changes to the information representation is key to achieving any desired behaviours.

Component design: ideally we have now broken down the application into a number of components whose functionality is well defined. We can now focus on developing each component independently. Dividing up complexity in this way is essential to avoid being overwhelmed by it. We may also start to see that some component functionality is reusable in other applications. This reusable functionality is the basis for the patterns presented throughout this book.

If the components are still too complex, then they are broken down into sub-components (and so on) until complexity is manageable. As before, each component needs to be clear on:

- What information is specific to the component.
- What the component does with this information (how it is manipulated).
- What needs to be communicated with other components (what is the interface with everything external to the component).

Flow charting: the details of the information manipulation operations need to be outlined. The format of this is often personal preference since building a single component according to specifications provided from previous steps is usually the task of a single programmer. Novice programmers benefit from representing relatively simple sequences of operations with flowcharts or similar representations. This needs to be a usable working document. Clarity in the order in which operations occur, and clearly identified choice (conditionals) and loops (iteration) are more important than creating a perfectly laid out diagram with prescribed occult symbols. This may need to be modified as you work on it, so a rough diagram on paper that can be scribbled over is useful.

More complex elements benefit from using patterns. If the component is an instance of one or more patterns, such as those provided throughout this book, then

just transcribe the structure from this book, and adapt them to the particular information and operations required by the component.

Scripting: the process of turning your understanding, as captured in the previous steps, into lines of program code should now be fairly mechanical. You may need to consult examples or reference material to get the syntax completely correct but this becomes easier with practice. You should be aware that most developers do not write programs in one session from top to bottom. Instead they will (or should) often follow a progressive refinement approach: creating the skeletons for each function that is required, outlining the key steps in each function (with comments), filling in the skeletons (e.g. just the block boundaries) of the statements used for each step, and then finally writing the detail.

In terms of ensuring that the program works, the developer identifies stages of development that are tested at regular intervals. Rather than writing a complete system, write a single function and some code to test to see if it works first. Then run it and verify that the function behaves as expected. A similar approach is used to test each line of code as you write it.

The test cases used are worth keeping around in case you might need to modify the component in future. You can then repeat the tests to make sure the modifications haven't changed anything that you didn't anticipate.

Compiling: most programs, even scripts, undergo some internal validation to ensure they correspond with the grammatical rules of the language before being transformed into a version that can run either within the virtual reality engine or standalone. Errors reported at this stage are usually the result of careless typing, and are corrected after deciphering the often cryptic messages that are provided. Practice will acquaint you with the most common of these messages and what

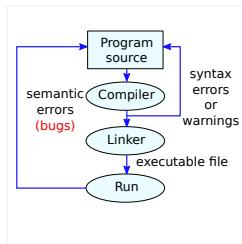


Figure 4.1.1: The process of turning program code into executable files involves several stages in which syntax errors, and semantic errors (bugs) need to be identified and resolved.

needs to be done to address them.

It is still worth paying attention to these messages as they can sometimes flag issues with the way in which you intend to use the programming language (e.g. forgetting to provide valid initial values to variables). Most systems provide various categories of message:

- Errors: issues that prevent the program from being run. These must be fixed in order to proceed.
- Warnings: the program is still able to run, but there are indications that the program may misbehave. The programmer is allowed to ignore these if they understand the implications of their decision.
- Log messages: these are messages provided by the program while it is running (and cannot be anticipated in advance). They often indicate that the program may have encountered an unexpected situation that it is not able to cope with, and can help diagnose the cause of the problem. One of the debugging patterns involves adding your own log messages to your programs so that you can diagnose issues with your programs.

The errors identified by the compiler relate to the *syntax* of your program. Despite the need to develop skills in interpreting the error messages provided, this process just ensures that your program conforms to the language conventions. The messages encountered at this stage are not bugs (logic errors in your reasoning), but can often hint at the existence of bugs.

Debugging: despite the best intentions, no program ever works perfectly the first time around. To identify bugs, you must:

- have a clear expectation of what the program (or part of the program) is expected to do. Based on your original design, be able to explain what the expected output should be given a particular input to the application.

- be able to reproduce the problem. If you can recreate the issue, then you are in a position to make changes to the program and verify whether the problem is solved.

Bugs are the result of issues with the *semantics* of the program. The logic of the instructions provided does not agree with the intent of the designer and programmer. Fixing bugs requires understanding the original design logic, matching that with the actual behaviour of the program, and modifying the algorithmic structure of the program where discrepancies exist.

4.2 *Printing*

4.2.1 *Description*

The main problem with trying to diagnose issues with your program's logic is the difficulty in seeing what is happening as the program runs. Since the behaviour in the virtual world depends on the changes of information (in the scene graph) and on the sequence of operations, we need to be able to view both values of variables and the order in which the program executes particular instructions. Viewing this information from within a virtual reality experience is difficult and the complexity of overlaying such information on the view of the virtual world can often introduce more issues than it helps diagnose.

Fortunately many systems still retain a link to a traditional text based log. This may be stored in a file on the system, or scrolled through in a text window. A single statement can usually be added anywhere in your program to generate a customized text message whenever the program executes that part of the program.

4.2.2 *Pattern*

The print statement is used for debugging by using this particular pattern:

```
print ("Statement identifier", information  
values)
```

The statement identifier is a unique identifier representing that particular debugging message. Care should be taken to make it unique within the application, to avoid misleading the programmer during debugging. This could be done by including the name of the file and function containing the print statement, along with a unique number identifying that particular print statement within the function. A combination of file number and line number within the file is another common convention, particularly where the programming environment supports automating this process.

The information values referred to are a list of relevant variable names. Relevant variables are those that are modified in the proximity of the print statement, and whose values are related to any issue that has been identified.

It is worth being thorough, putting in plentiful statements providing detailed information. Remember that bugs tend to lurk where “it is obvious” that no error exists. Bugs that lurk in plain sight are always corrected immediately and so are not the subject of any debugging process. It is the bugs that you cannot spot that need this level of intervention.

4.2.3 *Example*

Example 20. Applying the pattern to Unity software

1. The use of the print pattern is demonstrated in conjunction with creating a component to move an object up and down. The object must go up until it reaches an upper bound, and then start going down. When it hits a lower bound, it then starts going up again.

Create a new object, and add a C# script called MovingUpAndDown to it. Initially the script consists of the code in Algorithm 31 on the next page.

2. If we try and run this, we notice nothing is happening. We would have expected the object to actually move. Since there are no error messages in the console (if there are, then look for syntax errors in the code you wrote) we need to try to identify what is going wrong.

Include the print statement shown below in the first line of the update function. This allows us to validate that the update function is being invoked for this object. A novice programmer might expect to be able to assume this since this wise and authoritative book has already said that this function is invoked on every frame, but an experienced programmer will realise that testing even such obvious assumptions will help identify the problem faster.

```
1 print ("Update function running");
```

When we try to run the application now, we see ... nothing. No message printed to the console. Therefore the update function is not running. What could be wrong? Did we forget to add the script component to an object in the scene? Yes, we did. Fix that problem, and try again.

3. When we run the program after adding the C# script component to the object we should see messages being printed to the console tab as shown in Figure 4.2.1 on the following page. The object to which the script is attached also moves upwards, but then comes to a stop. We would have expected it to move up and then start moving down. This indicates that there is another bug in the code.
4. We'll add some more print statements to try to track down the issue. Since we're not seeing any downward movement, our hypothesis might be that the

Algorithm 31 A bug filled script intended to autonomously move objects up and down.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Move the object up and down between the bounds
6  // defined.
6  public class MovingUpAndDown : MonoBehaviour {
7
8      public float upperBound = 3;
9      public float lowerBound = -2;
10
11     private bool directionUp = true;
12
13     // Use this for initialization
14     void Start () {
15         }
16
17     // Update is called once per frame
18     void Update () {
19         if (this.transform.position.y > upperBound)
20         {
21             directionUp = false;
22             this.transform.position += new Vector3 (0, -0.1f
23                                         , 0);
24         }
25         if (this.transform.position.y < upperBound)
26         {
27             directionUp = true;
28             this.transform.position += new Vector3 (0, 0.1f,
29                                         0);
30         }
  
```

Project Console

```

083650 Update function running
UnityEngine.MonoBehaviourprintObject
  
```

Figure 4.2.1: The output of the print statement appears in the console window. When multiple duplicate messages are printed the counter on the right hand side is incremented.

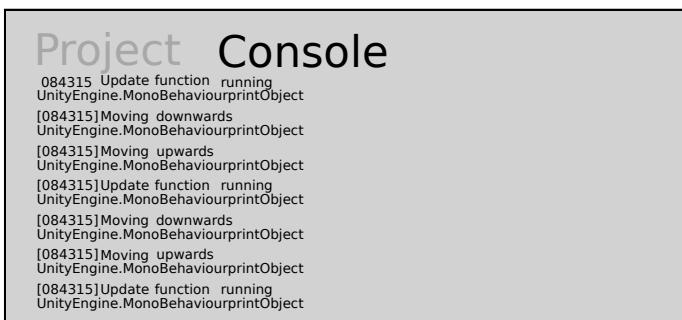
downward moving condition is never triggered. Let us put print statements into the body of each of the conditionals (if statements).

```

1   if (this.transform.position.y > upperBound)
2   {
3       print ("Moving downwards");
4       directionUp = false;
5       this.transform.position += new Vector3 (0, -0.1
6           f, 0);
7   }
8   if (this.transform.position.y < upperBound)
9   {
10      print ("Moving upwards");
11      directionUp = true;
12      this.transform.position += new Vector3 (0, 0.1f
13          , 0);
14  }

```

The results of this change appear in the console window as shown in Figure 4.2.2. The astute observer will notice that the “Moving upwards” messages appears over and over again when the application starts (and the object moves upwards), but then when the object stops *both* the “Moving upwards” *and* the “Moving downwards” messages are printed (and at the same rate that the “Update function running” message is printed). This suggests that both conditions are true



The screenshot shows the Unity Project Console window. The title bar says "Project Console". The console output area contains the following text:

```

084315 Update function running
UnityEngine.MonoBehaviourprintObject
[084315]Moving downwards
UnityEngine.MonoBehaviourprintObject
[084315]Moving upwards
UnityEngine.MonoBehaviourprintObject
[084315]Update function running
UnityEngine.MonoBehaviourprintObject
[084315]Moving downwards
UnityEngine.MonoBehaviourprintObject
[084315]Moving upwards
UnityEngine.MonoBehaviourprintObject
[084315]Update function running
UnityEngine.MonoBehaviourprintObject

```

Figure 4.2.2: The console output after additional print statements are added.

at the same time. If we check the logic in the program code we might spot our terrible mistake. When lazily copying the first condition to make the second the inequality operator was correctly reversed, but the limit was not updated. Moving upwards should

be triggered when reaching the *lower* bound, not the upper. Both conditions only use the upper bound at present.

5. Changing the second condition to use the variable lowerBound removes this bug. However the program still doesn't work. Reviewing the console output confirms that only the "Update function running" message is printed, so the update function is indeed being used. Neither of the conditions is true though, which is reasonable since the object is now both above the lower bound and below the upper bound.

This particular bug is identified in another example. In the meantime, can you spot the flaw? Alternatively, create your own short script to achieve the stated goals of this component. It may not always be possible to turn severely flawed code into a working solution.

4.3 *Watching in slow motion*

4.3.1 *Description*

Diagnosing problems in most programs is complicated by the inability to see what is actually happening inside the black box that the program is running on. More specifically the manipulation of information that programs do is largely invisible, apart from the occasional side effect such as a result being printed on the screen. Virtual reality systems are slightly better than most in that much more of the information state corresponds to information that is visible on the screen. Still this is not always enough to help diagnose an issue happening in a single function affecting just one object. It would be much better if we could actually see the program executing line by line and see the effect of each operation.

It turns out that we can.

Most development environments have some form of integrated debugger. This allows you to:

- Mark individual lines of program code as breakpoints. The program pauses when it reaches that line and allows the programmer to inspect the value of particular variables or step through the code one line at a time.
- Watch particular variable values. Since programs are all about modifying information, and that information affects the behaviour of the virtual world, we need to see what values are being used and how they are changing. You can mark particular variables as needing to be watched, after which their names and values appear in a separate window for you to view as you step through the code.
- You have various controls to step over lines of code. These include: step one line (including into functions), step over functions (useful if you trust the function, or if some else wrote it and you cannot change it), or continue running until the next time a break point is encountered.

How to use a debugger: Identify the line of code or function that you believe contains the error. If you follow the philosophy of testing your code regularly, then this is probably a piece of the program code that you've added or changed recently. Place a break point here. Once the program stops at this break point, think. What should the values of the properties be at this stage in the program's life? How will they change with each line? Then test your theories. If the results don't match your expectations - why not? Your theory may be wrong, in which case you may need to think about how you designed your solution. Or the program may be wrong, and you've identified the line that is misbehaving. If the mistake is obvious, fix it. If you don't understand why it is not working, rubber duck it.^{1,2}

Virtual reality development environments offer another form of live debugging with the ability to watch properties. Most environments include some form of property inspector which shows live values of the se-

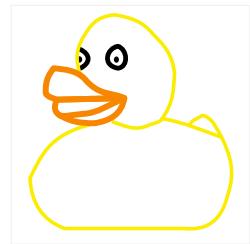


Figure 4.3.1: When the debugging seems impossible, explain your problem to another person or a rubber duck.

¹ Rubber ducking involves explaining the problem to another person, which is remarkably effective in identifying the issue even before the other person responds.

² If you don't have another person handy, any inanimate object will do. Even the rubber duck from your bath time rituals.

lected object's properties even when the application is running. Strategies to try here include: watch the scene graph. Where is your object? It may not be visible in the rendered image, but if it exists in the scene graph then you can check properties to see where it is, and if it is enabled or active. Watch the scene graph as well to diagnose issues where object creation may be getting out of control. Select a particular object and check the scene graph to verify that any parent-child hierarchy operations are working correctly. Select the object and check the list of components to ensure that all those expected are present and active. This includes any script components that should be there, since debugging the code for these is impossible if the code is never invoked.

Don't be surprised about how often impossible events occur or how common incredibly obvious mistakes are. Things are always more obvious once you know the answer. An experienced programmer is just someone who has made the mistakes often enough to have become familiar with the process. Be very suspicious of any program that works first time. It probably means that the bug is so insidious that its effects can't even be seen directly.³

³ Or the code might actually just be correct.

4.3.2 *Example*

Example 21. Applying the pattern to Unity software

1. In this example we're going to continue the debugging of some code developed in a previous example. The intention is to develop a component that causes an object to move up and down vertically, changing directions when it hits a particular upper bound height, or the lower bound height. The component as it stands at the moment is as defined in Algorithm 32 on the facing page, and is attached to an object in the scene. The previous debugging attempt has given us some insight. We know that the update function is being called, and that a previous error in one of the condi-

Algorithm 32 Initial attempt to produce a script that makes an object oscillate vertically between two defined height levels.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Move the object up and down between the bounds
   defined.
6  public class MovingUpAndDown : MonoBehaviour {
7
8      public float upperBound = 3;
9      public float lowerBound = -2;
10
11     private bool directionUp = true;
12
13     // Use this for initialization
14     void Start () {
15         }
16
17     // Update is called once per frame
18     void Update () {
19         if (this.transform.position.y > upperBound)
20         {
21             directionUp = false;
22             this.transform.position += new Vector3 (0, -0.1f
23                           , 0);
24         }
25         if (this.transform.position.y < lowerBound)
26         {
27             directionUp = true;
28             this.transform.position += new Vector3 (0, 0.1f,
29                           0);
30         }
}
```

tions has been resolved. Previously the object moved so we anticipate that the vectors being added to the position value have some effect, but since correcting the previous error nothing is moving.

2. We make use of the interactive debugger under Unity software to see what is actually happening. This assumes that MonoDevelop is being used as the default editor.

Inside the editor, right click on the column to the left of the line numbers and adjacent to the first statement in the Update function. Select “Toggle Breakpoint”. This should result in a marker being added to that column, and the line being highlighted as shown in Figure 4.3.2. A break point indicates a point where

Figure 4.3.2: Adding a break point to the first line of the update method.

MovingUpAndDown.cs

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Move the object up and down between the bounds defined.
6  public class MovingUpAndDown : MonoBehaviour {
7
8      public float upperBound = 3;
9      public float lowerBound = -2;
10
11     private bool directionUp = true;
12
13    // Use this for initialization
14    void Start () {
15
16    }
17
18    // Update is called once per frame
19    void Update () {
20        if (this.transform.position.y > upperBound)
21        {
22            directionUp = false;
23            this.transform.position += new Vector3 (0, -0.1f, 0);
24        }
25        if (this.transform.position.y < lowerBound)
26        {
27            directionUp = true;
28            this.transform.position += new Vector3 (0, 0.1f, 0);
29        }
30    }
31 }
```

the program will pause and allow you to probe more deeply into the workings of the code.

3. We now enable the debugging process. Inside MonoDevelop, select Run/Attach to Process and attach to the Unity software Editor process listed in the box that pops up. This allows the debugger to monitor the execution of applications that are run through the Unity software editor.

Now go to the Unity software editor, and press the play button to run the application. Once the application has started, you should notice that MonoDevelop asks for your attention: it has interrupted the program at the given break point and now gives you the chance to examine the program state, and to control the execution of the program from this point.

4. A portion of the debugging view within MonoDevelop is shown in Figure 4.3.3 on the following page. The Watch pane allows us to inspect and monitor the value of particular variables within the application. We expect that these variables change in value as the program manipulates the information stored in them and in the scene graph. Click in the window to add the names of variables to watch. Some suggestions on variables to examine include: *upperBound*, *lowerBound*, *directionUp* (the variables specific to this process), and *this.transform.position* (the element of the scene graph that is being referenced and modified). Note that the position variable is a Vector3 object and can be opened up in the watch window to see all its fields and properties as shown in Figure 4.3.4 on the next page.
5. The other thing we can do in the debugger is to step through the code one line at a time, and watch the effect of that line on the variables in the watch window. The *Run/Step Over* or *Run/Step Into* menu options facilitate this, although there are also buttons on the

```

MovingUpAndDown.cs

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Move the object up and down between the bounds defined.
6  public class MovingUpAndDown : MonoBehaviour {
7
8      public float upperBound = 3;
9      public float lowerBound = -2;
10
11     private bool directionUp = true;
12
13     // Use this for initialization
14     void Start () {
15
16     }
17
18     // Update is called once per frame
19     void Update () {
20         if (this.transform.position.y > upperBound)
21         {
22             directionUp = false;
23             this.transform.position += new Vector3 (0, -0.1f, 0);
24         }
25         if (this.transform.position.y < lowerBound)
26         {
27             directionUp = true;
28             this.transform.position += new Vector3 (0, 0.1f, 0);
29         }
30     }
31 }

```

Watch	Locals	Breakpoints	Threads

Figure 4.3.3: Debugger view once the program execution reaches the marked breakpoint. In particular, note the Watch pane which allows variable values to be examined.

Watch	Locals	Breakpoints	Threads
Name	Value		Type
upperBound	3		float
directionUp	true		bool
this.transform.position	0.0 0.00.0		UnityEngine.Vector3
magnitude	0		float
normalized	0.0 0.00.0		UnityEngine.Vector3
sqrMagnitude	0		float
x	0		float
y	0		float
z	0		float

Figure 4.3.4: The watch window in the debugger showing variable names and values.

control bar that perform the same actions. *Step Into* steps into any functions that are called, while *Step Over* moves to the next line of the current function.

Step over the lines in the Update function a few times. It quickly becomes apparent that neither of the conditions are fulfilled by an object in the middle of the allowed height range, and as a consequence, no change to the position is ever made.

The fix to this bug is to move the position change code outside of the conditional blocks, and make sure that movement occurs in each update step. The revised version in Algorithm 33 on the following page now functions correctly, and might be closer match to code that you would have designed yourself. Run the debugger over this version, and watch the interaction of object's position with the value in the *directionIncrement* variable.

4.4 *Reductio ad collapsum*

4.4.1 *Description*

Sometimes some bugs in your code are incredibly persistent and you get to the stage where you start to suspect the compiler is not interpreting your code correctly, the engine has an error in it, or the universe has determined that you shall never be a programmer. In practice, this is rarely the case. However this might be the point at which you want to get someone else to have a look at your problem, or where you might want to file a bug report with your engine manufacturer.

Don't just bundle up your entire project and email it off to someone else. Apart from leaking potentially sensitive information about the project that you're working on, nobody wants to try to pick up a complex semi-working project and figure out what is going on with it. There are conventions to reporting bugs, whether you're taking advantage of the person sitting next to you, rely-

Algorithm 33 Version of the vertical oscillator component with one less bug.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Move the object up and down between the bounds
6  // defined.
6  public class MovingUpAndDown : MonoBehaviour {
7
8      public float upperBound = 3;
9      public float lowerBound = -2;
10
11     private Vector3 directionIncrement = new Vector3 (0,
12             0.1f, 0);
13
14     // Use this for initialization
15     void Start () {
16
17         // Update is called once per frame
18         void Update () {
19             if (this.transform.position.y > upperBound)
20             {
21                 directionIncrement = new Vector3 (0, -0.1f, 0);
22             }
23             if (this.transform.position.y < lowerBound)
24             {
25                 directionIncrement = new Vector3 (0, 0.1f, 0);
26             }
27             this.transform.position += directionIncrement;
28         }
29     }
```

ing on the goodwill of strangers in an online forum, or expecting support from a vendor hotline.

The first step is to reduce your problem to the smallest example that still has the issue. Ideally start with a completely blank project. Add the bare minimum of content to the project to demonstrate the error. Don't add complex geometry (unless required). Keep to a variety of distinct (i.e. not all exactly the same) primitive elements so that when you explain that you're trying to make the white cube move there is only one white cube in the scene. Add in the minimal amount of scripting. Preferably only one file that seems to contain the problem. Even then remove any sections of code that are not relevant to recreating the issue. Even better, start from scratch and recreate the handful of lines needed to recreate the problem.

There is a good chance that you will spot the problem while doing this. The chance is even greater if you recreate the project from scratch rather than copying in material from the problem case. Recreating prevents invisible and accidental typos (such as O rather than o) from being simply perpetuated.

You will also meet with a more sympathetic attitude from those volunteering to help you. It is clear that you are putting in the effort to help yourself and not relying on others to do your homework for you.⁴

4.4.2 *Pattern*

The pattern for producing the smallest non-working example, and a meaningful bug report:

1. Identify clearly to yourself, and then later to others: what is the specific problem that I need a solution to. This should not be the overall problem of producing the specific virtual reality application that you're working on. Rather it is one specific task for one of the components identified in your design.
2. Start with a blank project. Write the shortest code

⁴ If in doubt, compare responses on Internet support forums to questions phrased either as:

- How do I write a program to do X?
- This is my program to do X but is not working because of (very clearly explained issue). What can I do to fix it?
- I am trying to do Y, in order to produce a program to do X. Here is a short code fragment that I'm using but it produces A instead of B. What can I do to fix it?

Which do you think would receive the most help? Interestingly, asking for help from your tutors can produce a similar outcome.

you can to solve the problem identified previously using the approach that is implemented in your non-working application. Make sure this project still exhibits the issue in a visible and repeatable fashion. Ideally hard code in any input, events or actions required to reproduce the problem. You want to make sure any other person running this project will see the issue immediately and without the need to read and follow a complex set of instructions to set up the right circumstances.

- (a) If your new version works, then compare with the non-working project.
 - i. If you spot the issue in the original project, fix it.
 - ii. If the problem remains in the original system, you may need to include more elements in the simplified example.
 - iii. If the simplified example has all of the functionality of the original and still works then you've solved the problem.
3. Make sure all functions, variables and scene objects are well named, and can easily be identified by a written description. Take screenshots showing the issue, and annotate them to clearly show which bit is not working (draw a big red ring around it).
4. Post up your bug report, with clear description of the specific process that is required, the sample project, instructions to use (if required), and a clear description of:
 - (a) how to tell if the process is exhibiting the problem (use the screenshots).
 - (b) what it should look like if the problem were solved.
5. Asking nicely for help is also a good thing. Do check back and respond if any responses are provided. Do

also consider the etiquette of the forum and acknowledge, offer thanks or rate answers once you have received help.

4.5 *Testing and Unit tests*

4.5.1 *Description*

You found a bug in your code and fixed it. Well done! Three months later you had to modify portion of a particular function and now your program is misbehaving again. Now it is a lot harder to debug because you've written more scripts in the meantime, and you've long since forgotten about how you built some of the older functions and what input they might have assumed.

The process of testing and quality assurance is hopefully something you're starting to do implicitly while you write your programs. You've realised that writing a complete program from top to bottom in a single session and getting it to run first time is an impossible and undesirable dream. You now write only one function, or a portion of a function, before compiling and running your application and testing that specific code.

This is still a reasonably ad-hoc process. You might add in a few lines of code to test your function with some common parameter values but will delete these lines once you're satisfied that everything is behaving correctly. We need to make this process more rigorous, and we need to have some way of revalidating a function if we have to modify it later.

The testing processes described in this pattern focus on testing individual functions that you develop. There are further testing processes required to ensure that your application works, once all of these functions are integrated into the application. For virtual reality systems, there are further testing stages that assess the user's response to the environment that is created to ensure that it is effective and usable.

4.5.2 *Pattern*

Function testing: consider a function that takes a number of parameters:

```
function myFunction (Type parameter1,
                     Type parameter2, ...)
{
    // body of function to be tested
    return result
}
```

We need to make sure that we get the correct result for any particular set of parameter values. Furthermore the function must also avoid failure scenarios before returning a result. Such failure can result from dependencies in the function of other subsystems; receiving a value from a controller, being able to find and open a file, or having enough storage in a collection to add a new object.

Testing a function thoroughly requires not only testing that it works when expected, but that it also behaves in a predictable way when given unexpected values. A professional tester tries to identify all of the edge cases and anticipate what scenarios the function programmer may have forgotten to consider. Future uses of the function may actually use these cases. It is easier to adapt the function when it is freshly written than doing it months later.

Typical edge cases:

- For integers: 0, very big numbers, negative numbers (particularly where the function may involve a square root operation), floating point numbers, any values that may trigger a divide by zero situation.
- For floating point values: 0, integers, negative numbers, anything that may involve comparison (two parameters with very similar values), anything that may trigger divide by zero, very large numbers (high exponents), anything that may trigger a rounding issue (very detailed numbers with long mantissae).

- For strings: empty string, null (empty reference), string with one letter, string with strange characters (other alphabets), string with special characters (control codes), strings with the string terminator character, non-terminated strings (depending on language), extremely long strings.⁵

Unit testing: Once a function has been tested with a thorough range of parameter values, it is a waste to discard this effort. Unit testing involves writing an additional function that tests another function. The code inside the unit test has a table of potential inputs, and their corresponding correct results. The unit test function iterates through each of these cases, invokes the function with the inputs, compares the result returned with the correct output, and finally prints out a tally of how many of these test cases passed. The details of any failure cases are also reported.

The application can then have an option to invoke all of the unit test functions across an entire application. This might be triggered only a daily basis by an automated server in the evenings once all work has been completed. The programmer will return the next day to a reassuring report indicating that none of the recent changes have had an adverse effect. Alternatively, any issues identified are quickly tracked down while the nature of recent program changes is still fresh in their minds.

Writing unit tests requires extra effort with the intent that this more than pays for itself in reduced bug-tracking time for large projects.

4.5.3 Example

Example 22. Applying the pattern to Unity software

1. This example demonstrates the use of unit tests, and a simplified test harness that could be applied in a Unity software project. The context is a particular class that we're creating to support an underwater swimming

⁵ "A QA engineer walks into a bar, and orders a beer. Then he orders o beers. Then he orders 99999999999 beers. Then he orders an lizard. Then he orders nothing. Then he orders -1 beers. Then he orders null beers. Then he orders aeutnhdtiu. Then he orders a "><script>drop tables;</script>. Finally, the QA engineer leaves without paying, comes back, and asks for the tab." Original source unknown.

with dolphins virtual reality experience. We need to make sure all objects stay in the water, without the risk of swimming above the surface, or through the sand at the bottom of the ocean. The class created to satisfy this need is shown in Algorithm 34.

2. We now create a second class; the test harness to perform some unit tests on the functions within the SeaCheck class to determine if it is working correctly. Create an empty object in the scene, called TestingStation. We attach all of the testing scripts to this object, allowing all the testing processes to be disabled by just disabling this object. This allows us to toggle easily between a testing and production version of our project. Create a new C# component called TestHarness which contains the test functions. We'll do all the testing once when the application starts, so add a call to our (to be created) test function to the Start function:

```

1    void Start () {
2        // Runs some unit tests.
3        // Disable the object in the scene to disable
4        // testing.
5        testSeaCheckClass ();
6    }

```

3. The test function creates instances of the SeaCheck class (with particular parameter values), and then make calls to the inSea method also with particular parameter values. It then compares the result to what is expected for those parameters, and prints out a success or failure message as appropriate. There is a large amount of duplicated code in the example shown in Algorithm 35 which could be factored down into a function that creates, tests and reports for a given set of parameters.
4. The results are shown in Figure 4.5.1. The value of unit tests has already paid off as one of these has failed, indicating a flaw in the function. We can then continue to reuse the testing process to validate if any

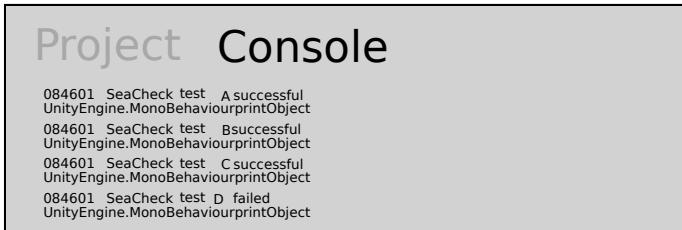
Algorithm 34 A class to manage sea levels and to provide a public interface that allows a given height value to be compared to the floor and surface levels of a body of water. This class needs to be tested to determine if it works as advertised.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class SeaCheck {
6
7      // Height of the sea floor
8      private int seaFloor;
9      // Height of the sea surface
10     private int seaSurface;
11
12    public SeaCheck (int floor, int surface)
13    {
14        seaFloor = floor;
15        seaSurface = surface;
16    }
17
18    // Checks to see if the height value provided as a
19    // parameter is in the sea i.e. between floor and
20    // surface.
21    public bool inSea (float height)
22    {
23        if (((int) height > seaFloor) && ((int) height <
24            seaSurface))
25        {
26            return true;
27        }
28        return false;
29    }
}
```

Algorithm 35 Some examples of unit tests that are applied to the SeaCheck class and methods.

```
1 void testSeaCheckClass ()
2 {
3     SeaCheck seaCheck;
4
5     seaCheck = new SeaCheck (-5, 5);
6     if (seaCheck.inSea (0.0f) == true)
7     {
8         print ("SeaCheck, test A successful");
9     }
10    else
11    {
12        print ("SeaCheck, test A failed");
13    }
14    if (seaCheck.inSea (-10.0f) == false)
15    {
16        print ("SeaCheck, test B successful");
17    }
18    else
19    {
20        print ("SeaCheck, test B failed");
21    }
22    if (seaCheck.inSea (10.0f) == false)
23    {
24        print ("SeaCheck, test C successful");
25    }
26    else
27    {
28        print ("SeaCheck, test C failed");
29    }
30
31    seaCheck = new SeaCheck (0, 1);
32    if (seaCheck.inSea (0.5f) == true)
33    {
34        print ("SeaCheck, test D successful");
35    }
36    else
37    {
38        print ("SeaCheck, test D failed");
39    }
40 }
```

changes that we make actually correct the issue, or if they may have inadvertently introduced problems to code that was previously working. It is a good idea



The screenshot shows a terminal window with a light gray background and a dark gray header bar. The header bar contains the text "Project Console" in a large, bold, black font. Below the header, the terminal output is displayed in a monospaced black font. The output consists of five lines of text, each starting with "084601 SeaCheck test". The first four lines are labeled "A", "B", "C", and "D" respectively, followed by the word "successful" and the path "UnityEngine MonoBehaviourprintObject". The fifth line is labeled "D" and followed by the word "failed" and the same path.

```
084601 SeaCheck test A successful
UnityEngine	MonoBehaviourprintObject
084601 SeaCheck test Bsuccessful
UnityEngine	MonoBehaviourprintObject
084601 SeaCheck test C successful
UnityEngine	MonoBehaviourprintObject
084601 SeaCheck test D failed
UnityEngine	MonoBehaviourprintObject
```

Figure 4.5.1: Results of the unit testing process.

to include additional tests to cover cases that are not yet tested (for example, what should happen if we call `SeaCheck (5, -5)`? Any problems encountered while using the `SeaCheck` class in future would also suggest examples of unit tests that could be applied to ensure that similar situations do not arise in future.

5

Responsive Worlds

5.1 Event Handler

5.1.1 Description

In other programming environments the event handler pattern is usually integrated with other patterns that register and invoke it. Since most of this complexity has been integrated into the virtual reality engine, we confine ourselves to just the aspect related to responding to events that occur in the virtual world.

As background, consider how objects in a virtual world interact with one another. There are a few obvious options:

- Each object knows about every other object and calls specific functions associated with components on the other object when some interaction is required. This requires a large number of potential functions that must be supported, every object must be aware of every other object, and the application designer must have considered all likely scenarios right at the start.
- Each object can call a single function on every other object at regular intervals to check to see if that object has any specific message for it. This does reduce the number of potential functions that need to be supported and trades this off for a more extensible set of messages that might need to be communicated. It is

still inefficient though because most of the time there are no messages passed between any pair of objects.

The event handler mechanism aims to improve on these options. Each object defines (registers) a set of functions that represent the messages (events) that it accepts. Each such function is then an event handler that defines that object's response to a particular event. Part of the registration process includes the condition under which those messages are relevant. These conditions can include options such as: send event when object is created, send an event every frame, send an event when another object is in close proximity (or colliding), send an event after a period of time or when receiving input from controller devices or other virtual reality hardware peripherals. This process is extended to include custom events that individual objects may send to all others (broadcast), those nearby, or those visible in a particular direction.

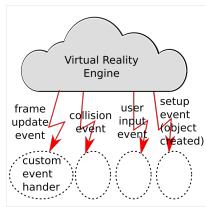


Figure 5.1.1: Extending the behaviour of a virtual reality application involves scripting handlers for events produced by the virtual reality engine in response to changes in the virtual world.

Many virtual reality engines expect each object to provide event handlers for set up (when the object is created), and update (on each frame) events. The object can then define extra event handlers for virtual reality engine standard events such as collisions and timer events which must adhere to particular format requirements defined by the engine. Additional functions are created for application or object specific events but the developer is then also responsible for generating these events and ensuring that the object is registered to receive such events.

5.1.2 Pattern

Typically an event handler pattern takes the form:

```
function eventHandler (event ev)
{
    ...
    ... (script defining response)
}
```

The name of the function (eventHandler in this case) needs to be replaced with the appropriate name defined

by the virtual reality engine's event mechanism. The developer may only choose their own names for custom events that they create for an object. Even then it is wise to adhere to a common convention (defined by the engine or standardized across the development team) since these names represent part of the interface that the object exposes to other components of the virtual reality application.

The parameters passed to the event handler are also often specific to the event being handled. These must also be exactly as defined by the virtual reality engine's documentation or there is a risk that the function won't be identified as an event handler. Some systems may not pass all information to the event handler via function parameters but can expect the function to query global application state to get some of this information.

The body of the function should contain code that represents how the object responds to a particular event (e.g. it may change colour when a collision event is detected). Ideally it should only modify its own properties. Direct interference with other objects runs the risk of introducing dependencies between objects that hampers reuse and extension of your application. If reaction from other objects is required it is advisable to rather send them an event, and let them manage their response to that.

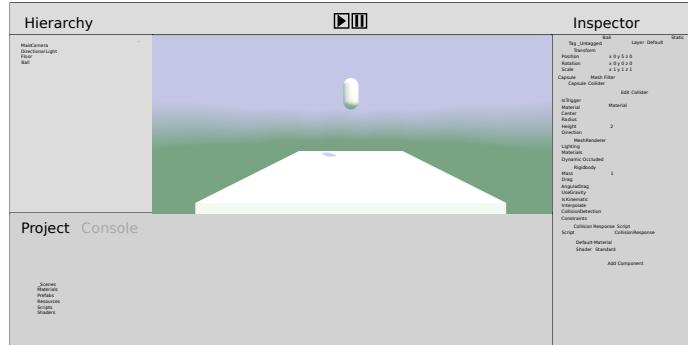
5.1.3 Example

Example 23. Applying the pattern to Unity software

1. This example demonstrates a collision event handler, one of the more common event handlers supported by the virtual reality engines. We use this to create a bouncing ball. The ball object falls under the simulated force of gravity, and receive an upwards impulse (in the form of an instantaneous upwards velocity value) whenever it hits another object so that it bounces up again.
2. Create a scene with an object to represent the ground.

A large flattened cube suffices for this purpose. Now create the ball, and position it about the ground object. Add a Physics/Rigidbody component to this (leaving the gravity checkbox turned on). If you run the simulation at this stage, the ball should fall to the ground and remain there. The scene layout is illustrated in Figure 5.1.2. Add a new CollisionResponse C# script

Figure 5.1.2: The ball and plane configuration to create a bouncing ball by adding a response to the collision event.



to the project and attach this to the ball. The collision event handler in Unity software is named `OnCollisionEnter`. We'll set this up to deliver an upward velocity to ball whenever a collision occurs, by setting the velocity property of the rigid body component. Also, just for fun, we change the colour of the other object to demonstrate how to access parameters of the collision from the parameter provided to the collision event handler. The code below is included in the CollisionResponse class.

```

1 void OnCollisionEnter (Collision collision)
2 {
3     this.GetComponent <Rigidbody> ().velocity = new
4         Vector3 (0, 10, 0);
5     collision.gameObject.GetComponent <MeshRenderer>
6         ().material.color = new Color (Random.Range
7             (0.0f, 1.0f), Random.Range (0.0f, 1.0f),
8             Random.Range (0.0f, 1.0f));
9 }
```

5.2 Keeping track of time

5.2.1 Description

Activities in virtual reality applications are sensitive to time. Usage of time related information falls into the following categories:

1. What is the current time?
2. How much time has passed since the last time we performed this operation?
3. How can I wait for a certain amount of time to pass?

The current time is determined from the real-time clock that is a hardware component of most computers. The virtual reality engine usually defines a global Time object that provides various functions to retrieve this information. The precision with which time is measured depends on the particular source of the timing information. Clocks that are specific to the nearest second, or hundredth of a second often provide access to a date value as well which helps track long duration activities. Other high speed timers may report time in micro- or nanoseconds allowing considerable precision but these values may start to repeat or cycle every few minutes or hours. However the difference between two successive readings of a high speed timer reports the time that has elapsed, provided that less than one cycle has passed between the two readings.

The amount of time that it takes to return to the same point in the program is generally a useful value. For example, if simulating the movement of an object travelling with a constant velocity, the position of the object is updated in discrete steps by repeating incrementing it by the same amount over and over. The size of the step (amount to move the object) depends on how often the operation is repeated and is directly proportional to the time that has elapsed since the last update was

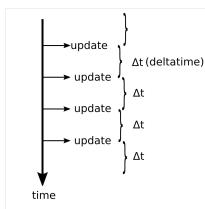


Figure 5.2.1: The change in time between successive occurrences of the same event is Δt (deltaTime). For common events this may be provided as a parameter or system global variable, but can also be calculated for any custom event handlers.

made. Some event handlers explicitly provide a parameter `deltaTime` representing the time since the last event (others may provide `deltaTime` as a global variable). The greek symbol delta (Δ) is used to refer to an increment or change in a particular value.

If a `deltaTime` variable is not available, then this process can be performed manually. The program must use a variable to store the time at which it was last at this point in the program, and subtract this value from the current time to work out how much time has elapsed.

Some applications may require actions to occur at a particular time, or after a certain amount of time has elapsed. It is possible to get the program to perform the equivalent of clock-watching: doing nothing while waiting for time to pass. This is rather inefficient use of the processing resource which could be better used in continuing to provide a responsive and interactive virtual reality experience.

A preferred approach is to schedule an event that is generated at some point in the future. This requires use of particular facilities in the programming language or virtual reality engine. These facilities may include timer objects that generate events, or access to lower level functionality to create a separate thread of execution (allowing part of the program to run concurrently (at the same time - sort of) as the main application). This thread then goes to sleep for the prescribed duration, before waking and either triggering the intended event or just perform the required action itself.

5.2.2 *Pattern*

Retrieving the current time involves access to the system clock, usually involving just a single call to a system function:

```
Time time = Clock.getTime ()
```

The type of the `time` variable depends on the nature of the functions available. These can include `DateTime`

types, which would have fields related to day, month, year, hour, minute and seconds. Alternatively high speed counters would return integers (measuring time in micro- or nano-seconds) or floating point values (probably using units of seconds).

The per-frame event handlers usually have access to the time that has elapsed since the last frame:

```
function update (deltaTime)
{
    // alternative if parameter not provided.
    deltaTime = System.deltaTime;
}
```

To create your own way of measuring elapsed time since last reaching a particular point in a program we define a global variable that tracks the last time that point was reached. Depending on language features this variable may be better achieved as a static variable, or a private class variable as it only needs to be available at one point in the program but does need to have its value persist between each iteration.

```
Time lastTimeAtPointX;
...
function X ()
{
    Time currentTimeAtPointX = Clock.getTime ();
    Time deltaTime = currentTimeAtPointX - lastTimeAtPointX;
    lastTimeAtPointX = currentTimeAtPointX;
    ...
}
```

Once we have used our stored value to calculate deltaTime, we update the stored value so that it has the appropriate value for the next iteration.

Triggering an action at some time in the future can use different but equivalent patterns depending on facilities provided by the programming environment.

If there are Timer objects with the ability to generate events, then an event is triggered in t seconds time using:

```

// code to request an event in  $t$  seconds time.
Timer.scheduleEvent (timeEventHandler,  $t$ );
...
function timeEventHandler ()
{
    // do whatever is required after  $t$  seconds.
}
...

```

This pattern consists of two key elements: an event handler which responds to the event at the desired point in the future, and the scheduling of the event which needs to know when the event is produced (in t seconds) and what function is triggered when the time has elapsed. The name of the function is passed in as a parameter when scheduling the event.

The alternative approach relies on separate threads of execution. Once again, an action takes place t seconds in the future.

```

// code to create a concurrent thread
// of execution
t = Thread ();
t.start ();
...
class Thread
{
    function run ()
    {
        sleep ( $t$ );
        // do whatever is required after  $t$  seconds.
        ...
    }
}

```

The run function of the Thread executes concurrently with the flow of processing of the main application. The thread function then immediate suspends itself by going

to sleep so that it consumes no resources until time t has passed. After that it gets the opportunity to continue running and perform the delayed action.

5.2.3 Example

Example 24. Applying the pattern to Unity software

1. This example demonstrates how to schedule an action to take place at some time in the future. A coroutine is created which represents an extra thread of execution that continues concurrently with the current one. Specifically this means that your application can continue running without having to wait for the time to pass, while the coroutine can put itself to sleep without consuming any processing resource for a given delay period, after which it can resume.

The sample provided just changes the colour of the object to which the script is attached. The colour change is triggered by a key press, but only takes place a specified number of seconds after that action. Create a new object in the scene and attach the script from Algorithm 36 to it.

2. The process starts in the Update function. The Unity software input subsystem is queried to check if a particular key is pressed (the one associated with the Fire1 axis, which by default includes the left control key, or the left mouse button). Rather than make an immediate call to the DelayAndChange function, this function call is wrapped in the StartCoroutine function which ensures that this function call takes place in a what appears as a separate thread of execution.
3. Coroutines have a particular form. They have a return type of IEnumerator which allows them to be interrupted and resume at a later time. The yield statement is used to surrender control, with the understanding that execution resumes at some time later from the next

Algorithm 36 A coroutine is used to trigger an action at some point in the future.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class DelayedResponse : MonoBehaviour {
6
7      public float delayTime = 5.0f;
8
9      private bool coRoutineRunning = false;
10
11     // Use this for initialization
12     void Start () {
13     }
14
15     IEnumerator DelayAndChange (float delayPeriod)
16     {
17         if (!coRoutineRunning)
18         {
19             coRoutineRunning = true;
20             yield return new WaitForSeconds (delayPeriod);
21             Color oldColour = this.GetComponent <
22                 MeshRenderer> ().materials[0].color;
23             this.GetComponent <MeshRenderer> ().materials
24                 [0].color = new Color (1, 0, 1);
25             yield return new WaitForSeconds (0.5f);
26             this.GetComponent <MeshRenderer> ().materials
27                 [0].color = oldColour;
28             coRoutineRunning = false;
29         }
30     }
31
32     // Update is called once per frame
33     void Update () {
34         if (Input.GetAxis ("Fire1") > 0)
35         {
36             StartCoroutine (DelayAndChange (delayTime));
37         }
38     }
39 }
```

line. The `WaitForSeconds` function specifies the delay before resuming.

4. As an extra enhancement for this example, the coroutine delays for a period of time, changes the colour of the objects, delays for a small extra period of time and restores the colour. It copies the previous material colour to a local variable. This version does demonstrate some of the unexpected effects that can result from the use of concurrency. If the key is pressed multiple times (try disabling the `if (!coRoutineRunning)` line to see this) multiple coroutines are created during subsequent frames.

These later versions assume that the colour of the object as changed by a previous occurrence of the routine is in fact its original colour, and dutifully restore the object to this colour when they exit. This overrides the colour restoration provided by the first occurrence of the coroutine, and so the object briefly flashes to its original colour before being “corrected”.

The solution to this is to make sure that only one coroutine can run at a time. Since coroutines share the variables that are global to the class, we have included a Boolean flag that the first coroutine sets, and that is only cleared when that coroutine ends. Any other coroutine created during this time is not allowed to make any changes to object colour and terminates immediately.

5.3 *Taking control*

5.3.1 *Description*

Ideally the virtual reality engine should manage the input that the user provides through the various virtual reality controller devices that may be attached. This input system could provide facilities such as:

- abstracting over the devices available so that the ap-

plication could concentrate on what actions are being performed, rather than controller specific details such as which button or joystick is being used. The actions would be defined by the application (e.g. pick up, select, activate) and a separate mapping table maintained to relate this to the controls available on each category of controller.

- sending events when particular input is provided so that the application doesn't have to keep checking. The alternative of checking regularly for input is less efficient but if required should ensure that we don't have to check every device individually while doing so.

Engine support for this pattern has been limited, potentially due to the range of different devices that are being introduced. In the interim the pattern below sketches out the structure of an input abstraction layer that could be implemented and reused across a number of virtual reality applications.

5.3.2 *Pattern*

The input abstraction layer maps the individual controls on a range of different input devices into actions supported by the application. So to start off, we need a list of the different types of controls available:

```
enum ControlTypes {Button, 2DRange, 3DLocation}
```

Here anything that produces a click/press type action is classified as a button, and controls such as touchpads and joysticks that provide an x and y pair within a finite range would fall into the 2DRange category. A controller that is tracked in space might provide a 3DLocation. This list is not meant to be exhaustive and other devices may fall into additional categories. The goal though, is to ensure that we can find some way of satisfying the input

needs of an application regardless of the types of controls available.

We also need a list of the input actions that the application supports. For simplicity we provide this as a list of functions that perform these actions in the application. These are event handlers for the action events.

```
actionHandlers = [ SelectPressed,
                   ControllerMoved,
                   ActivatePressed,
                   MenuMoved ]
```

These actions are application specific, and typically vary between applications. The actions are chosen so that they correspond to particular types of control that are defined.

We now define a table that relates each action to the control type involved, and the individual controller control that provides it. Updating the controls for the application should only require updating this table.

```
controlMapping =
{
    SelectionPressed : Button :
        [MouseLeftClicked,
         ControllerTriggerPressed ],
    ControllerMoved : 3DLocation :
        [ControllerTracking,
         KeyboardMovementEmulator ],
    ActivatePressed : Button :
        [MouseRightClicked,
         ControllerGripPressed,
         ControllerTouchpadClicked ],
    MenuMoved : 2DRange :
        [MouseTracking,
         ControllerTouchpadMove ]
}
```

Each of the action handlers is matched up to a list of different controllers and individual controls within those

controllers. The control types provided are used to ensure that meaningless control options (e.g. driving a 3DLocation input with a Button control) are screened out.

The remainder of the pattern now becomes independent of the specific application. This is also the only part of the pattern that needs to engage directly with the idiosyncrasies of individual hardware devices so protects the application from having to manage those concerns.

```
// A frame event handler on a input
// manager object
function update ()
{
    for each action in controlMapping
    {
        for each control in controlMapping[action]
        {

            if input available on this control
            {

                invoke action event handler
                with this input as parameter
            }
        }
    }
}
```

The action event handler is invoked directly by calling the function provided. Alternatively, if the virtual reality engine allows the application to create custom events, then these could be dispatched as events. Calling event handlers directly imposes potential delays in the input management process if the application takes a long time to process input, and we might want to avoid jitter in responding to potentially time critical input.

5.3.3 Example

Example 25. Applying the pattern to Unity software

1. This example demonstrates a portion of the input manager applied to the keyboard and mouse input devices. Both of these are already accessible through Unity's built-in input manager but this example does show how the pattern could be extended to other devices, or customized to support any novel forms of input required by a particular application.

This pattern has also deliberately been kept simple. This means that it does not achieve the full efficiencies and reconfigurability that could be achieved through a thorough but more complex implementation of this pattern. The ambitious might like to consider extensions such as: internally inverting the control type to device source mapping to achieve faster lookups, including different types of callback event handling that take parameters specific to the control type, or including support for additional input devices (through polling other data sources in the Update method).

2. Based on the pattern above, we create an input manager that supports some simple user movement. In this case the user can move forwards/backwards, and turn left/right. Initially we expect this could be achieved using arrow and other common navigation keys on the keyboard, and by using the mouse position. In future this might be extended to using a joystick or touchpad on a virtual reality controller, or the tilt of a rotational controller.

As per the pattern we define three sets of constants representing the various levels of abstraction. The control types refer to the class of information provided. This might be boolean values such as a button click, or a value in the range $[-1, 1]$ from a slider or joystick. Note that Unity software already remaps movement keys into such a range (+1 for the forward key, -1 for

the backward key).

```
1 public enum ControlTypes { ButtonAction,
    AxisAction };
```

The action handlers represent the abstraction required by the virtual reality application. Our current application is very limited and only needs to respond to a forward/backward movement direction, and a left/right sideways angle.

```
1 public enum ActionHandlers { MoveForward,
    TurnSideway };
```

Finally the actual hardware sources now need to be identified (we're treating the Unity software input service as a hardware source for the purpose of this example since we're overriding it rather than integrating with it). This needs to be broken down to the level of source of each individual control input that provide one of the actions.

```
1 enum DeviceSources { UnityInputHorizontal,
    UnityInputVertical, UnityInputMouseX,
    UnityInputMouseY, ViveControllerTrigger };
```

The Vive trigger is not currently used but included to illustrate where additional hardware sources would be included.

3. We now keep a table indicating which devices supply each control/action pair. Each row in this table consists of one control value, its associated action value and a list of devices that can supply that action. This is the significant aspect of this pattern. We can support extra sources of input, or customise existing ones just by changing the entries in this list of devices. Each row also keep a list of event handlers; these identify the objects in the virtual world that are interested in values from that particular input source. Typically we would expect one controller to only control one object in the virtual world so there would usually only be one handler per row.

One row of this table is represented using a class that groups all of these elements.

```

1  class ControlMappingElement
2  {
3      public ControlTypes controlType;
4      public ActionHandlers action;
5      public List<DeviceSources> devices;
6      public List<System.Action <float>> eventHandlers;
7
8      public ControlMappingElement (ControlTypes c,
9          ActionHandlers a, List<DeviceSources> l)
10     {
11         action = a;
12         controlType = c;
13         devices = l;
14         eventHandlers = new List<System.Action <float>>
15             ();
16     }
17 };

```

We now define the specific table that maps input to actions for our specific application. This table would be the only structure modified when we need to configure which devices control particular forms of input.

```

1  private ControlMappingElement [] controlMap = new
2      ControlMappingElement [] {
3          new ControlMappingElement (ControlTypes.
4              AxisAction, ActionHandlers.
5              MoveForward, new List<DeviceSources>
6                  { DeviceSources.UnityInputVertical,
7                      DeviceSources.UnityInputMouseY }),
8          new ControlMappingElement (ControlTypes.
9              AxisAction, ActionHandlers.
10             TurnSideway, new List<DeviceSources>
11                 { DeviceSources.UnityInputHorizontal,
12                     DeviceSources.UnityInputMouseX }),
13      };

```

4. This table does not predefine the event handlers interested in each action. These are configured from the external objects directly, by calling a function in our input manager that we make publicly accessible. The external object specifies (as parameters to our function) which control type and action they are interested and provide a function of their own that we call when we have relevant input for them. Note the external

users of our input manager service have no need to know about which particular hardware device needs to be used to provide that action.

```

1  public void registerForInput (ControlTypes c,
      ActionHandlers a, System.Action <float>
      eventHandler)
2  {
3      foreach (ControlMappingElement cme in controlMap)
4      {
5          if ((cme.controlType == c) && (cme.action == a)
6              )
7          {
8              cme.eventHandlers.Add (eventHandler);
9          }
10     }

```

This function searches the table until it finds rows matching the control type and action, and then appends the provided event handler to the list associated with that action.

5. The mechanics of actually collecting input from the various sources is performed in the Update function for the input manager. The sources that we use all require regularly polling to check if new input has arrived. Ideally we would also include event handlers so we wouldn't have to repeatedly check for input from devices that provide less frequent updates.

This example polls only a single source, but additional calls to check for input from other devices could easily be added.

```

1  void Update () {
2      pollUnityInput ();
3      // ... and poll anything else required.
4  }

```

The polling process is relegated to a separate function because these now become specific to the process of retrieving input from a particular hardware source. Retrieving input from the (virtual) Unity software input source uses:

```

1 void pollUnityInput ()
2 {
3     float h = Input.GetAxis ("Horizontal");
4     registerInput (DeviceSources.UnityInputHorizontal,
5                     h);
6     float v = Input.GetAxis ("Vertical");
7     registerInput (DeviceSources.UnityInputVertical,
8                     v);
9     float mx = Input.GetAxis ("Mouse X");
10    registerInput (DeviceSources.UnityInputMouseX, mx
11                    );
11 }
```

- Once any input is received it is passed to a function that identifies which source is interested in it, and dispatch the information. This could be a completely asynchronous event dispatching system, or the somewhat more direct approach shown below that makes direct calls to the event handlers for any action that takes input from a particular source.

```

1 void registerInput (DeviceSources source, float
                      value)
2 {
3     foreach (ControlMappingElement cme in controlMap)
4     {
5         foreach (DeviceSources ds in cme.devices)
6         {
7             if (ds == source)
8             {
9                 foreach (System.Action <float> ev in cme.
10                     eventHandlers)
11                 {
12                     ev (value);
13                 }
14             }
15         }
16     }
```

The multiple nested loops used here are less efficient than they could be. We could potentially create a second table when the application starts that provides a direct mapping from a particular device source to a list of event handlers by scanning through the rows in

controlMap. Users of the input manager would still only need to work with the more application centric view provided when initializing controlMap.

7. An external virtual reality object using the input manager only needs to register an event handler when they need a particular form of input (specified as action, not individual device). Assuming this object had a variable inputSystem referencing the input manager:

```
1  inputSystem.registerForInput (InputSystem.
    ControlTypes.AxisAction, InputSystem.
    ActionHandlers.MoveForward, forwardEventHandler
);
```

The event handler function can then happily perform the required action appropriate to the object when input corresponding to that action is received, regardless of whether it originates at keyboard, mouse, or controller.

```
1  void forwardEventHandler (float v)
2  {
3      this.transform.position = this.transform.position
        + speed * v * this.transform.forward;
4 }
```

5.4 Locomotion in a particular direction

5.4.1 Description

The controllers and the head mounted display in a virtual world are tracked so that their position and orientation properties are regularly updated. Gaze or controller directed locomotion can use the direction indicated by one of these tracked objects to support locomotion through the virtual world. This form of locomotion is not recommended for use in any non-trivial virtual reality application due to the sudden and non-physically relevant accelerations experienced by the user. Rather consider locomotion patterns such as teleportation (section 5.6 on page 198)

5.4.2 Pattern

Given a tracked object in the virtual world identified by the variable *directionObject*, and the object representing the user's avatar (containing the camera) in the variable *userObject* the pattern for locomotion is:

```
function update (deltaTime)
{
    if (controller.triggerPressed ())
    {
        userObject.position = userObject.position +
            speed * directionObject.forward * deltaTime
    }
}
```

The movement is controlled by the trigger on the controller. No movement occurs until the trigger is pressed. Once the trigger is pressed movement starts abruptly and continues until the trigger is released.

This pattern does have a speed property which is set to a suitable value when the application is initialized. The use of deltaTime means that movement occurs at a consistent speed, independent of the update rate of the application.

This pattern takes advantage of one of the alternative interpretations of orientation (section 2.5 on page 43) that uses orientation represented as three vectors corresponding to up, sideways and forward. We want our locomotion to be in the forward direction of the controller.

An alternative pattern with more gradual acceleration and deceleration is shown below. This may be less jerky, but also takes longer to respond to trigger presses and releases.

```
function update (deltaTime)
{
```

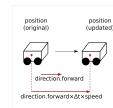


Figure 5.4.1: Simple kinematic motion is achieved by adding a direction vector to the current position. The time since the last position update is used to adjust the distance moved in a single step.

```

        if (controller.triggerPressed () and
            speed < speedMax)
        {

            speed = speed + acceleration * deltaTime
        }
        else
        {

            speed = speed - acceleration * deltaTime
        }
        userObject.position = userObject.position +
            speed * directionObject.forward * deltaTime
    }
}

```

This pattern requires extra constants representing the magnitude of acceleration, and the maximum speed.

5.4.3 Example

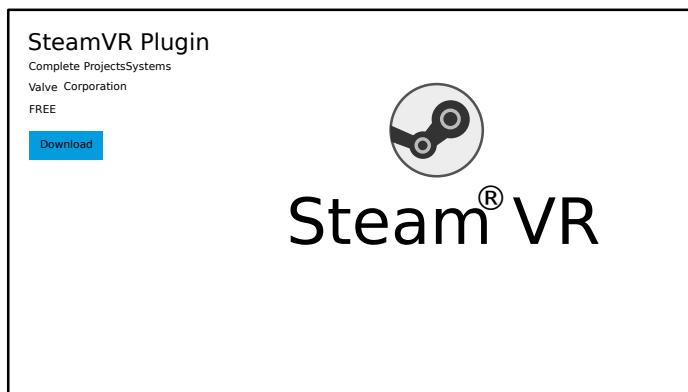
Example 26. Applying the pattern to Unity software

1. This pattern allows motion in a direction defined by another object in the virtual world. Typically this direction would be taken from a tracked head mounted display or virtual reality controller. The steps below have been tested with the HTC Vive, but should be adaptable to many of the other systems available.

Start a new Unity software project. To include the facilities required by the Vive, switch to the Asset Store view, and find, download and add the SteamVR Plugin¹ to the project (Figure 5.4.2). The assets for this are usually added to a folder named SteamVR. We keep a separate folder for third party assets called "Standard Assets", and recommend moving the SteamVR folder into this. From the Standard Assets/SteamVR/Prefabs folder, add the CameraRig and SteamVR prefabs to the scene. The existing Main Camera should be deleted

¹ ©2018 Valve Corporation. Steam and the Steam logo are trademarks and/or registered trademarks of Valve Corporation in the U.S. and/or other countries.

Figure 5.4.2: The StreamVR plugin provides prefabs to support particular head mounted displays and controllers.



or disabled. Running the application should produce an (empty) environment responding to movement of the head mounted display, and should also display the controllers when they are within the viewing area as shown in Figure 5.4.3.



Figure 5.4.3: Controllers should be visible in the virtual world when placed in front of the user in the physical world.

2. This example adapts the Unity software Input Manager settings to include axes that respond to various controller buttons.² Under Edit/Project Settings/Input, open the input manager. In the Inspector pane, open the Axes list and add a new axis to respond to controller trigger input. Name this LeftTrigger, select the Type as Joystick axis, and select axis 9. Do the same for RightTrigger, but select axis 10, as shown in

² The axis values are based on the table for various virtual reality controllers provided at: <https://docs.unity3d.com/Manual/OpenVRControllers.html>

Figure 5.4.4.

Code such as that below now reads the controller triggers.

```
1 float v1 = Input.GetAxis ("LeftTrigger");
2 float v2 = Input.GetAxis ("RightTrigger");
3 print ("Triggers: " + v1 + " " + v2);
```

Note: The SteamVR input system is being updated to use its own input manager pattern (see section 5.3 on page 179). At the time this section was last updated, this change was underway. Input is no longer being provided as raw controller values, but instead control input is mapped to particular actions. This requires a variable to define which action is required.

```
1 public SteamVR_Action_Boolean trigger;
```

This variable can be assigned one of the actions resulting from trigger input, such as GrabPinch. Reading the trigger is then done through the following code:

```
1 float v1 = trigger.GetState (SteamVR_Input_Sources.
    LeftHand) ? 1 : 0;
2 float v2 = trigger.GetState (SteamVR_Input_Sources.
    RightHand) ? 1 : 0;
```

Once the new system stabilizes, we will stop referring to individual controls (such as triggers) and just use the actions provided by this input manager.

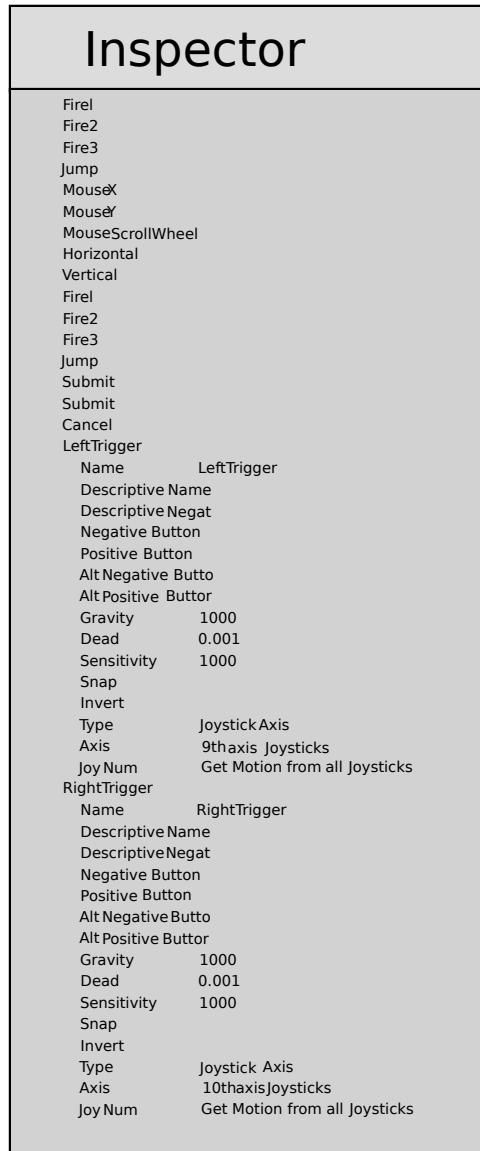
3. Create a new C# script component called MoveInThe-DirectionOfTrackedObject. This provides one key public property - the particular controller (head, left or right) whose orientation defines the direction of movement. A second property allows the speed of movement to be modified.

```
1 public GameObject steeringObject;
2 private float speed = 3.0f;
```

The update function for this component consists of:

```
1 void Update () {
```

Figure 5.4.4: Input axis settings for axes that respond to controller trigger input.



```

2     float v1 = Input.GetAxis ("LeftTrigger") + (
3         trigger.GetState (SteamVR_Input_Sources.
4             LeftHand) ? 1 : 0);
5     float v2 = Input.GetAxis ("RightTrigger") + (
6         trigger.GetState (SteamVR_Input_Sources.
7             RightHand) ? 1 : 0);
8     if ((v1 > 0) || (v2 > 0)) {
9         this.transform.position = this.transform.
10            position + speed * steeringObject.transform.
11            forward * Time.deltaTime;
12    }
13 }
```

Create an empty object called “Steering Experience”. Add 3 copies of the MoveInDirectionOfTrackedObject C# script component to it. Set Camera (Eye), Controller (left) and Controller (right) objects from the CameraRig as the values of SteeringObject for each of the 3 copies of MoveInDirectionOfTrackedObject. Disable each of these 3 components. Make the CameraRig a child of this steering experience object, so that the view moves with it. The steering experience object effectively becomes the body of the avatar that is being controlled by the locomotion pattern. The complete script is shown in Algorithm 37 on the next page.

4. Add a ground plane with a texture image on it to provide a visual reference. This helps you identify the direction of movement. Run the application. Enable one of the MoveInDirectionOfTrackerObject components at a time, as illustrated in Figure 5.4.5. When either of the controller triggers is pressed, the user’s avatar moves in the direction of the enabled tracked device.

5.5 Point and select

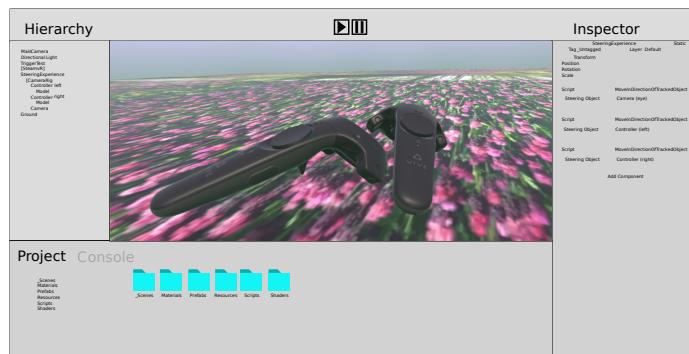
5.5.1 Description

Selecting objects within reach is achieved by adding a collision event handler to either the controller or the object. Selecting objects that are out of reach is achieved

Algorithm 37 This script captures the trigger input from the controller, although also incorporates elements of future usage that will make use of actions provided by an input manager.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Valve.VR;
5
6  public class MoveInDirectionOfTrackedObject :
    MonoBehaviour {
7
8      // Suggest setting this to gesture: GrabPinch
9      public SteamVR_Action_Boolean trigger;
10
11     public GameObject steeringObject;
12
13     private float speed = 3.0f;
14
15     // Use this for initialization
16     void Start () {
17     }
18
19     // Update is called once per frame
20     void Update () {
21         float v1 = Input.GetAxis ("LeftTrigger") + (
22             trigger.GetState (SteamVR_Input_Sources.
23                 LeftHand) ? 1 : 0);
24         float v2 = Input.GetAxis ("RightTrigger") + (
25             trigger.GetState (SteamVR_Input_Sources.
26                 RightHand) ? 1 : 0);
27         if ((v1 > 0) || (v2 > 0)) {
28             this.transform.position = this.transform.
29                 position + speed * steeringObject.transform.
30                 forward * Time.deltaTime;
31         }
32     }
33 }
```

Figure 5.4.5: Scene structure for controller/head based movement in the direction of the tracked object.



by aiming the controller in the direction of the target, firing a laser beam and selecting any object that the beam strikes.

In practice we don't actually emit laser beams although we may use such a representation to visually represent the process to the user. Internally we employ a process known as raycasting. A ray is half a line; it starts at a particular point and proceeds forever in a single direction (whereas the line passes through the point in both the forward and reverse directions). A ray is thus defined by two key properties:

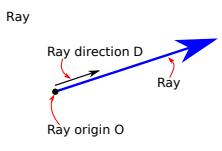


Figure 5.5.1: A ray is defined by its starting point (origin) and its direction provided as a vector.

- The origin of the ray, O . This is the point it is emitted from.
- The direction of the ray, D . This is a vector (usually unit length) indicating the direction that the ray follows.

Most engines support a raycasting operation that, when provided with a ray, will identify the closest object that the ray intersects. Sometimes a bound is specified which requires objects not to be too far away, no further than this bound. If the ray doesn't touch any objects, or if the intersection point is beyond the specified bound, then a null result is returned.

In practice, with a controller based raycasting solution, the position of the controller is the origin of the ray and

the forward direction for the controller is the direction of the ray.

5.5.2 Pattern

This pattern calls a `selectObject` function whenever the ray from the controller touches an object (alternatively, a `selectObject` event could be generated). The selection process is started by pressing the trigger on the controller.

```
function update (deltaTime)
{
    rayLimit = limit
    if (controller.triggerPressed ())
    {
        (hitObject, distance) =
            RayCast (Ray (controller.position,
                           controller.forward), rayLimit)
        if (hitObject != null)
        {
            rayLimit = distance
            selectObject (hitObject)
        }
        show laser beam (from controller.position
                        in direction controller.forward for
                        distance rayLimit)
    }
}
```

5.5.3 Example

Example 27. Applying the pattern to Unity software

1. This example allows a tracked object to be used as a pointer. It places a marker in the scene wherever the resulting ray strikes an object.
2. Import the SteamVR plugin from the Asset Store. Add the CameraRig and SteamVR components to the scene.

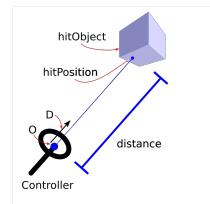


Figure 5.5.2: The raycasting operation is used for selection of objects in the virtual world, with the position and direction of a controller as origin and direction for the ray. The operation can return: the object intersected by the ray, the specific point of intersection, and the distance along the ray to this intersection.

Add in the left and right trigger axes to the input manager. Under Edit/Project Settings/Input, open the input manager. In the Inspector pane, open the Axes list and add a new axis to respond to controller trigger input. Name this LeftTrigger, select the Type as Joystick axis, and select axis 9. Do the same for RightTrigger, but select axis 10.

3. Create any elements required in the scene. This might include a ground plane, but also any objects that we want to point at with the ray casting.

Create a small spherical prefab. This is used as marker point for where the ray hits. It would help to disable the collider on this so that rays do not collide with it.

Create an empty, called SelectionManager. Attach a C# script component called FireRayFromDevice, as shown in Algorithm 38. Provide one of the controllers, and the target marker prefab as parameters.

4. Raycasting behaviour is triggered when one of the controller triggers is pressed. The marker prefab that is instantiated when the application starts, is placed at the position where the ray from the controller, in the direction that the controller is pointing, intersects an object in the scene as shown in Figure 5.5.3. The marker is left at that point until a new valid point is selected. Note that no intersection occurs between the ray and empty space, so no marker movement occurs when the controller not aimed at an actual object in the scene.

5.6 *Teleportation*

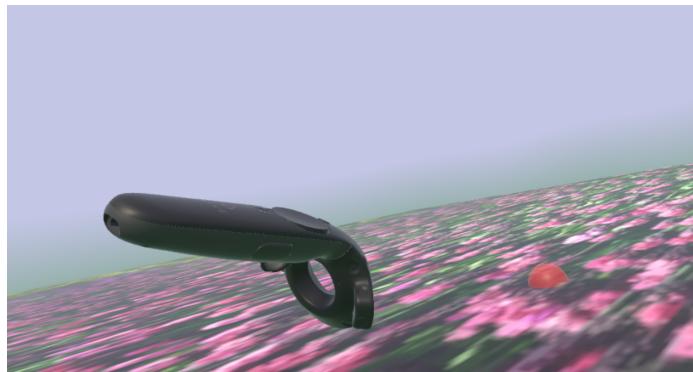
5.6.1 *Description*

Teleporting is a preferred method of locomotion. This is achieved by utilizing some previous patterns: use a raycast based selection to nominate a destination point (section 5.5 on page 194), check that the target point is

Algorithm 38 Pointing at an object, and selecting a point in space is demonstrated through the use of raycasting in this code example. The RaycastHit object has other fields that may be useful for related tasks.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Valve.VR;
5
6  public class FireRayFromDevice : MonoBehaviour {
7      // Suggest setting this to gesture: GrabPinch
8      public SteamVR_Action_Boolean trigger;
9      public GameObject controller;
10     public GameObject targetPrefab;
11     public GameObject avatar;
12
13     private GameObject targetMarker;
14     private Vector3 lastRayPoint;
15
16     void Start () {
17         targetMarker = Instantiate (targetPrefab);
18         targetMarker.SetActive (false);
19     }
20
21     void Update () {
22         // distance to search along ray.
23         float distance = 100.0f;
24
25         float v1 = Input.GetAxis ("LeftTrigger") + (
26             trigger.GetState (SteamVR_Input_Sources.
27                 LeftHand) ? 1 : 0);
28         float v2 = Input.GetAxis ("RightTrigger") + (
29             trigger.GetState (SteamVR_Input_Sources.
30                 RightHand) ? 1 : 0);
31         if ((v1 > 0) || (v2 > 0)) {
32             RaycastHit hit;
33             if (Physics.Raycast (new Ray (controller.
34                 transform.position, controller.transform.
35                 forward), out hit, distance)) {
36                 lastRayPoint = hit.point;
37                 targetMarker.transform.position = lastRayPoint
38                 ;
39                 targetMarker.SetActive (true);
40             }
41         }
42     }
43 }
```

Figure 5.5.3: A marker is shown placed at the point of intersection of the ray from the controller, and the ground plane.



valid, and then use another button to initiate the movement.

We use a variation of the ray cast process that not only returns the object hit by the ray, but also defines the exact position on the object that was hit. We can then update the position of the avatar to match this position, with potentially any offset required to ensure the player's head remains a body height above the surface. We don't want to teleport and appear to be partially embedded in the destination.

Checking the validity of the target point is optional but we might want to constrain teleportation to only end up on surfaces of a particular types (i.e. floor is good, other player's controller is bad). We can add label properties to objects to enforce this constraint.

5.6.2 Pattern

The pattern for teleportation is:

```
global selectedObject = null
global selectedPosition

// event handler for object selection event.
function selectObject (hitObject, hitPosition)
{
    selectedObject = hitObject
```

```

selectedPosition = hitPosition
place teleportation target on selectedObject

}

function update (deltaTime)
{
    ...
    if (selectedObject != null and
        controller.teleportButtonPressed ())
    {

        if (selectedObject.label contains "teleportable")
        {

            userObject.position = selectedPosition // + offset?
            clear teleportation target on selectedObject
            selectedObject = null
        }
    }
}
}

```

As part of the feedback to the user, this pattern includes the option to add some form of visible marker to the destination point, once it has been selected but before the teleport operation has been triggered.

5.6.3 Example

Example 28. Applying the pattern to Unity software

1. This example employs concepts applied from previous examples on pointing at, and selecting elements in the scene. An extra controller button is then used to teleport the user's avatar to the last selected location.
2. Import the SteamVR plugin from the Asset Store. Add the CameraRig and SteamVR components to the scene. Add in the left and right trigger axes to the input

manager. Under Edit/Project Settings/Input, open the input manager. In the Inspector pane, open the Axes list and add a new axis to respond to controller trigger input. Name this LeftTrigger, select the Type as Joystick axis, and select axis 9. Do the same for RightTrigger, but select axis 10. Add Left and Right Grip entries to the input manager. These use joystick axis 11 and 12. Note: The script provided also makes use of the actions from the SteamVR input manager. The grip variable should be set to a grip based action, such as GrabGrip.

3. Add an avatar object to the scene, and make the CameraRig a child of this object. Attach a C# component using the code from Algorithms 39 and 40 to achieve teleportation. The avatar object must be provided as a parameter to this script. Teleportation is achieved by setting the avatar's position to that of the selected point. A vertical offset is included to ensure the head is above this point for better visibility. This assumes the avatar was constructed with the head at its origin. This change could be avoided by constructing the avatar with the origin at the foot position.

Algorithm 39 Teleportation achieved through a relatively small extension to the point and select process (part 1).

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Valve.VR;
5
6  public class TeleportWithRay: MonoBehaviour {
7      // Suggest setting this to gesture: GrabPinch
8      public SteamVR_Action_Boolean trigger;
9      // Suggest setting this to gesture: GrabGrip
10     public SteamVR_Action_Boolean grip;
11
12    public GameObject controller;
13    public GameObject targetPrefab;
14    public GameObject avatar;
15
16    private GameObject targetMarker;
17    private Vector3 lastRayPoint;
18
19    void Start () {
20        targetMarker = Instantiate (targetPrefab);
21        targetMarker.SetActive (false);
22    }
23
24    void Update () {
25        // distance to search along ray.
26        float distance = 100.0f;
27        float v1 = Input.GetAxis ("LeftTrigger") + (
28            trigger.GetState (SteamVR_Input_Sources.
29            LeftHand) ? 1 : 0);
30        float v2 = Input.GetAxis ("RightTrigger") + (
31            trigger.GetState (SteamVR_Input_Sources.
32            RightHand) ? 1 : 0);
33        if ((v1 > 0) || (v2 > 0)) {
34            RaycastHit hit;
35            if (Physics.Raycast (new Ray (controller.
36                transform.position, controller.transform.
37                forward), out hit, distance)) {
38                lastRayPoint = hit.point;
39                targetMarker.transform.position = lastRayPoint
40                ;
41                targetMarker.SetActive (true);
42            }
43        }
44        ...
45    }

```

Algorithm 40 Teleportation achieved through a relatively small extension to the point and select process (part 2).

```
1      ...
2      float v3 = Input.GetAxis ("LeftGrip") + (grip.
3          GetState (SteamVR_Input_Sources.LeftHand) ? 1
4              : 0);
5      float v4 = Input.GetAxis ("RightGrip") + (grip.
6          GetState (SteamVR_Input_Sources.RightHand) ?
7              1 : 0);
8      if ((v3 > 0) || (v4 > 0)) {
9          avatar.transform.position = lastRayPoint + new
10             Vector3 (0, 1, 0);
11     }
```

6

A Working Virtual World

6.1 State machine

6.1.1 Description

Most objects in a virtual reality application are autonomous, in the sense that they need to sense, reason and act with respect to the (virtual) world around them. The users themselves also broadly fall into this category with the exception that an external biological processing component handles the reasoning stage. Even then it is worth having some form of artificial control for user avatars that can deal with cases when no human is available to fill a particular role in the virtual world.

Extending the code controlling the object becomes increasingly difficult as the behaviour for the object becomes more complex and with increases in the range of events that the object responds to. The state machine pattern provides a convenient level of abstraction which helps with understanding, debugging and extending this behaviour, defines a representation for defining and documenting behaviours, and integrates with event passing facilities provided in most virtual reality engines. Other contents also use state machines, such as for managing the interactions with a user interface, or with progressing a virtual reality application through a sequence of scenes.

The state of an object is all the information that defines

the current condition of an object. Think of this as all of the information required to recreate that object exactly as it was in the virtual world when recovering from a power failure. Game players are familiar with state as this is the information restored when reloading a game from a saved level. The state of an object consists typically of all the variables local to that object. In practice, with state machines, we may limit our state machine to use only the portion of the state relevant to defining different types of behaviour.

The state machine pattern reasons about the behaviour of objects based on the states that it may be in. Events that happen to the object may change its state. A graph (collection of nodes and edges) represents the state machine visually, where each node receives a name associated with the state. Each node has arrows (labelled with an event name) connecting to other nodes (states) indicating that the object transitions to the destination state if that event occurs. This representation is referred to as a state transition diagram.

Some events do not have any effect when the object is in a particular state. In those cases, there are no edges from that state's node labelled with that event.

6.1.2 Pattern

Start with a state transition diagram corresponding to the behaviour of an object. This is transformed into script elements as follows.

We defined variables to represent the state of the object. The only variables needed are those that are relevant to the state transitions covered by the diagram. In the interests of readability we use an enumerated type so that the program code corresponds directly with the state transition diagram.

```
enumeration state = { State1, State2,
    ... , StateN }
```

Ideally you use more meaningful state names than 1, 2

and N! Use of letters as shown in the following discussion is also poor practice. Name your states and events according to their roles and actions they represent.

The enumeration defines a variable type which is assigned only one of these symbolic state names at a time. Internally the state variable is usually represented as an integer and the development environment translates the symbolic names into unique numbers. It is also possible to use a string variable to represent the state and define each of the state names as constant strings, but this is rather inefficient when the program runs.

We then need to ensure that state transitions occur when events happen. Assuming we have an event handler Event1Handler, we check to see which state we are in, and issue a change state request if that state transitions to another when Event1 occurs. This involves translating the state transition diagram directly into the program code (see diagram in Figure 6.1.1).

```
function EventMHandler (event ev)
{
    switch (state):
        case stateA: changeState (stateH); break;
        case stateB: changeState (stateC); break;
        case stateF: changeState (stateU); break;
        case stateY: changeState (stateE); break;
        default: do nothing; break;
}
```

This outline assumes that on EventM, the state transition diagram shows that state A transitions to state H, that state B transitions to state C, that state F transitions to state U, that state Y transitions to state E, and that all other states are unaffected by this event.

Repeat this process for all events listed in the state transition diagram.

You can create the function changeState yourself, to update the state variable, and handle any operations that

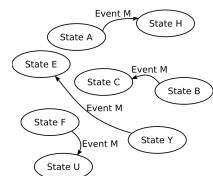


Figure 6.1.1: The portion of the state transition diagram corresponding to the EventM handler pattern shown.

may be needed when you have a change of state (perhaps updates to visual aspects of the object).

```
function changeState (newState)
{
    oldState = state
    state = newState
    // The code below is optional.
    sendEvent
        (stateChanged (oldState, newState),
         self)
}
```

The stateChanged event must not be one of the events in the state transition diagram or your program may just become interesting.

As an alternative (or in addition) to behaviour associated with change of states, it is also possible to have specific code that runs throughout the period that the object is in a particular state. This might animate a walking motion while a StateWalk is active, or periodically modify the size of the object in StateThrobbing.

This makes use of the event handler for the frame updates, along the lines of:

```
function update (deltaTime)
{
    switch (state):

        case StateA: do stateA actions; break;
        case StateB: do stateB actions; break;
        case StateC: do stateC actions; break;
        case StateD: do stateD actions; break;

}
```

The actions performed are specific to the nature of the object and its states. Some of these may involve incremental changes to position, orientation, scale or colour;

variables that are thus implicitly part of the state of the object. Most frame-update event handlers have access to a parameter or variable representing the amount of time that has passed since the last update. This is used to modulate the extent of any motion so that any actions occur at a consistent speed.

6.1.3 Example

Example 29. Applying the pattern to Unity software

1. This example discusses the creation of a component to produce a character (a lawyer) for participation in a virtual reality application that provides a virtual moot court experience. Moot courts allow student lawyers to practice their arguments and to experience the legal process from the perspective of the various participants in a trial. While the moot court is intended as a multi-participant virtual world (section 8.1 on page 265), there may not be enough human participants to fill all the roles and so the virtual reality application would control some of the avatars directly. The state machine pattern is suited to cases such as this where the autonomous avatar needs to perform a set sequence of actions, but where these actions must interact with events triggered by other participants, or activities in the virtual world.
2. The first step in applying the state machine pattern is to draw the state transition diagram. The actual implementation as program code proceeds mechanically from this, so all conceptual and design work should be done using this diagram. Trying to hack the code into shape independently of this diagram may succeed for simple cases, but produces an error prone and unmaintainable mess for anything of reasonable complexity. The diagram for the actions of the lawyer are as shown in Figure 6.1.2. The actions that need to take place during a state are shown in red. This state transition diagram is unusual in that it progresses

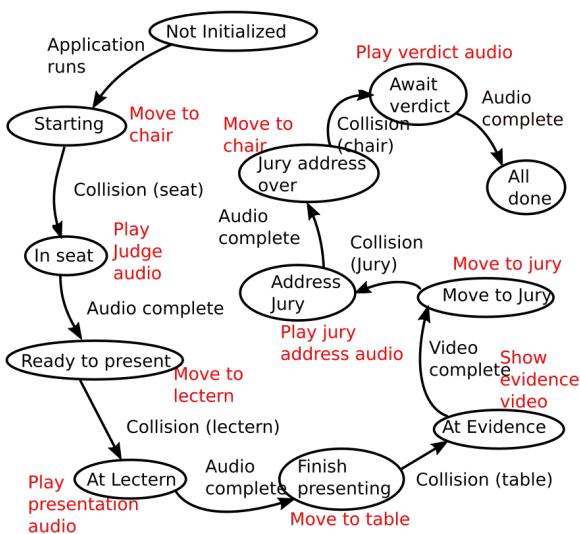


Figure 6.1.2: The states and events that the lawyer uses during the trial.

sequentially through each state in turn. Some of the other examples in this book present diagrams where several events may occur at any given state, and where previous states are revisited throughout the lifetime of the autonomous entity.

3. Within the program code, the states are represented with an enumerated type. Internally these are represented as specific integer constants, but the ability to write meaningful names in our program code allows the state transition diagram to be easily matched to the program structure.

```

1 // States of the lawyer.
2 enum LawyerState { NoStateSet, Starting,
                    InSeatGettingReady, ReadyToPresent,
                    PresentingAtLectern, ReadyForEvidence, Evidence
                    , MoveToJury, AddressJury, JuryAddressOver,
                    WaitingVerdict, AllDone };
  
```

An extra `NoStateSet` state is included, so we can represent a condition where the state variable has not

been formally initialized and the state machine has not yet been started.

4. The events in the state transition diagram are translated into program code for each of the event handlers.

Where ever the state transition diagram has an edge Y from node A to node B, then the event handler for event Y has a line of the form: if (state == A) then setState(B). The event handlers in this example are the onTriggerEnter function when the lawyer object collides with a collision box for its current destination, giveInstructions which plays a pre-prepared sound file (a legal argument) and changes to state B when done, and showMovie which plays a video file (presenting evidence) and changes to state B when done.

The only action that happens during a state is movement. Playing of sound and video is taken care of asynchronously by the virtual reality engine, so we don't need to keep advancing these directly in the update function. The update function conforms to the outline provided by the pattern but, since it performs movement for every state, it doesn't need to have the per state action as used in the pattern. States where the lawyer is stationary just set the destination to the current position, so the lawyer ends up taking zero length steps.

5. The setState function updates the state variable as is done in the state machine pattern. It generates a stateChanged event by calling the event handler for this event directly. The response to a state change from node A to node B on event Y is handled by a construct of the form: if (newState == B) then set destination position as the position associated with B, set the instructions as instructions associated with B, and play an media associated with state B.

Note that the actions in a state depend only on that state, and have no dependence on the previous states that lead to this state. Incorporating memory into

a state machine is generally undesirable but can be achieved by taking advantage of the oldState parameter in this example. The state transition diagram representation does not support notation that indicates prior states. Typically such systems are translated into a memoryless state transition diagram by having multiple different versions of the same state in the state transition diagram. Each version corresponds to a different path to reach the state. This can lead to state transition diagrams with large numbers of states.

6. The resulting program code to implement this particular state transition diagram is shown in Algorithms 41, 42, 43, 44, 45, 46 , 47 and 48.
7. To demonstrate the autonomous lawyer, create a new scene. Populate it with the following elements:
 - A ground plane.
 - Marker objects for key positions of the lawyer: chair, lectern, evidence table, and jury box.
 - A lawyer. Attach the LawyerBehaviour script to this object. Make sure the lawyer has a collider component defined. Add a number of Audio Source components to the lawyer, and attach suitable audio clips corresponding to the judge giving instructions, the lawyer's argument, the presentation to the jury and the judge's verdict.
 - A UI/Text object to hold the instructions that are shown on the screen to indicate the purposes of the lawyer's current state.
 - A semi-transparent cube of approximately the proportions of the lawyer object, to designate the point that the lawyer is moving towards, and to provide the object to collide with. This should be a prefab. Add a "TargetPoint" tag to this. Add a Rigidbody, and switch gravity off, so that it is able to register collision events. Enable the trigger checkbox in the collider component.

Algorithm 41 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 1).

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5
6  // As a lawyer, the user must complete activities:
7  // - move to seat. Until judge ready.
8  // - move to lectern, present. Until complete.
9  // - move to evidence table, present evidence.
10 // - move to jury, address jury. Until complete.
11 // - move to seat. Until verdict delivered.
12 // - task complete.
13 public class LawyerBehaviour : MonoBehaviour {
14     // States of the lawyer.
15     enum LawyerState { NoStateSet, Starting,
16                         InSeatGettingReady, ReadyToPresent,
17                         PresentingAtLectern, ReadyForEvidence, Evidence
18                         , MoveToJury, AddressJury, JuryAddressOver,
19                         WaitingVerdict, AllDone };
20
21     // Current state in the state transition graph.
22     private LawyerState state = LawyerState.NoStateSet;
23     // An object used to visually show the next point
24     // that has to be visited.
25     private GameObject targetPoint;
26     // External values required.
27     // A text object to hold instructions being provided
28     // to the user.
29     public Text instructionText;
30     // A prefab object to represent points of interest.
31     public GameObject targetPointMarkerPrefab;
32     // Positions of key elements to be visited.
33     public GameObject chairPosition;
34     public GameObject lecternPosition;
35     public GameObject evidenceTablePosition;
36     public GameObject juryPosition;
37     // The movie texture played when presenting evidence
38     public GameObject screenMovie;
39     // The sound clips for the various states.
40     public AudioSource judgeInstructions;
41     public AudioSource legalArgument;
42     public AudioSource juryPresentation;
43     public AudioSource judgeVerdict;
44
45     private float speedWhenAutonomous = 1.0f;

```

Algorithm 42 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 2).

```

1   // This function updates the lawyer simulation
      according to the state involved.
2   void setState (LawyerState newState)
3   {
4       if (state == newState)
5       {
6           return; // nothing changed.
7       }
8
9       LawyerState oldState = state;
10      state = newState;
11
12      handleStateChangedEvent (oldState, newState);
13  }
```

- Add a plane in the background to act as a movie screen. Create a material for this, and add a video clip as a texture (currently possible by importing a video file as a MovieTexture).

The public variables of the LawyerBehaviour component need to be filled in. The position fields need the chair, lectern, evidence table and jury box objects so that the lawyer can extract their positions. The instruction text field requires the UI/Text object created to show instructions. This screen object must also be used to set the corresponding property of the lawyer object.

Once running the lawyer moves from one location to the next, in response to the actions associated with each state in the Update function. Reaching a destination and colliding with the target marker triggers events to change to a subsequent state. Entering these states may trigger an audio or video playback, which in turns produces an event causing a state change when playback is complete. An example of simple Unity software scene containing the required elements is shown in Figure 6.1.3 on page 219.

Algorithm 43 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 3).

```

1   // Make any changes to the scene corresponding to
      arriving in the given state. The previous state
      is provided, but usually not required.
2   void handleStateChangedEvent (LawyerState oldState,
      LawyerState state)
3   {
4       if (state == LawyerState.Starting) {
5           targetPoint.transform.position = chairPosition
              .transform.position;
6           targetPoint.SetActive (true);
7           instructionText.text = "Make your way to your
              chair.";
8       }
9       else if (state == LawyerState.InSeatGettingReady
          ) {
10          targetPoint.SetActive (false);
11          instructionText.text = "Wait for the judge.";
12          StartCoroutine (giveInstructions (
              judgeInstructions, LawyerState.
              ReadyToPresent));
13      }
14      else if (state == LawyerState.ReadyToPresent) {
15          targetPoint.transform.position =
              lecternPosition.transform.position;
16          targetPoint.SetActive (true);
17          instructionText.text = "Move to the lectern.";
18      }
19      else if (state == LawyerState.
          PresentingAtLectern) {
20          targetPoint.SetActive (false);
21          instructionText.text = "Argue your legal
              position.";
22          StartCoroutine (giveInstructions (
              legalArgument, LawyerState.
              ReadyForEvidence));
23      }
24      else if (state == LawyerState.ReadyForEvidence)
          {
25          targetPoint.transform.position =
              evidenceTablePosition.transform.position;
26          targetPoint.SetActive (true);
27          instructionText.text = "Move to the evidence
              table.";
28      }
29      // continued...

```

Algorithm 44 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 4).

```

1      // continuation...
2      else if (state == LawyerState.Evidence) {
3          targetPoint.SetActive (false);
4          instructionText.text = "Show evidence.";
5          StartCoroutine (showMovie (screenMovie,
6                           LawyerState.MoveToJury));
7      }
8      else if (state == LawyerState.MoveToJury) {
9          targetPoint.transform.position = juryPosition.
10         transform.position;
11         instructionText.text = "Address the jury.";
12         targetPoint.SetActive (true);
13     }
14     else if (state == LawyerState.AddressJury) {
15         targetPoint.SetActive (false);
16         instructionText.text = "Summarize your case.";
17         StartCoroutine (giveInstructions (
18             juryPresentation, LawyerState.
19             JuryAddressOver));
20     }
21     else if (state == LawyerState.JuryAddressOver) {
22         targetPoint.transform.position = chairPosition
23         .transform.position;
24         instructionText.text = "Return to your chair."
25         ;
26         targetPoint.SetActive (true);
27     }
28     else if (state == LawyerState.WaitingVerdict) {
29         targetPoint.SetActive (false);
30         instructionText.text = "Judge presents
31         findings.";
32         StartCoroutine (giveInstructions (judgeVerdict
33             , LawyerState.AllDone));
34     }
35     else if (state == LawyerState.AllDone) {
36         targetPoint.SetActive (false);
37         instructionText.text = "The trial is now over.
38         ";
39     }
40 }
```

Algorithm 45 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 5).

```
1   // All event changes resulting from collisions. This
2   // is the collisionEventHandler function.
3   void OnTriggerEnter (Collider other)
4   {
5       if (other.tag == "TargetPoint")
6       {
7           if (state == LawyerState.Starting)
8           {
9               // in seat.
10              setState (LawyerState.InSeatGettingReady);
11          }
12          else if (state == LawyerState.ReadyToPresent)
13          {
14              // at lectern.
15              setState (LawyerState.PresentingAtLectern)
16              ;
17          }
18          else if (state == LawyerState.ReadyForEvidence
19                  )
20          {
21              // at evidence table.
22              setState (LawyerState.Evidence);
23          }
24          else if (state == LawyerState.MoveToJury)
25          {
26              // at jury.
27              setState (LawyerState.AddressJury);
28          }
29          else if (state == LawyerState.JuryAddressOver)
30          {
31              // back at chair.
32              setState (LawyerState.WaitingVerdict);
33          }
34      }
35  }
```

Algorithm 46 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 6).

```

1   // Play an audio file, and produce an event to
      change to the particular state once complete.
2   // This combines generating the audioComplete event
      and responding to it by setting the
3   // target state.
4   IEnumerator giveInstructions (


---



```

Algorithm 47 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 7).

```

1   // Play an movie file, and produce an event to
      change to the particular state once complete.
      This combines generating the videoComplete
      event and responding to it by setting the
      target state.
2   IEnumerator showMovie (GameObject movieObject,
                           LawyerState finalState)
3   {
4       MovieTexture movieTex = movieObject.GetComponent
                           <Renderer> ().material.mainTexture as
                           MovieTexture;
5       movieTex.Play ();
6       while (movieTex.isPlaying)
7       {
8           yield return new WaitForSeconds (1);
9       }
10      // End of presentation event occurs here.
11      setState (finalState);
12 }
```

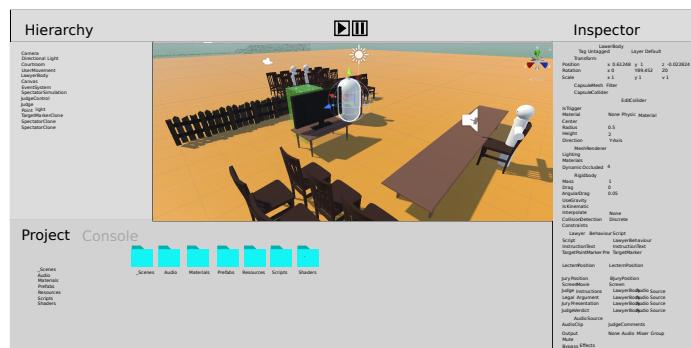
Algorithm 48 The state machine based on the state transition diagram for the autonomous lawyer in the moot court simulation (part 8).

```

1   void Start () {
2       targetPoint = Instantiate (targetPointMarkerPrefab
3                                     );
4       targetPoint.SetActive (false);
5       setState (LawyerState.Starting);
6   }
7
8   void Update () {
9
10    // handle any updates specific to the current
11    // state.
12    switch (state)
13    {
14        // do this for all states.
15        default:
16        {
17            Vector3 dir = targetPoint.transform.position -
18                transform.position;
19            if (dir.magnitude >= 1)
20            {
21                dir = dir / dir.magnitude;
22            }
23            transform.position += Time.deltaTime *
24                speedWhenAutonomous * dir;
25            transform.forward = dir;
26        }
27    }
}

```

Figure 6.1.3: A frame from the moot court sequence where the lawyer object in state Starting is heading towards the chair.



6.2 Autonomous agents

6.2.1 Description

Virtual worlds are more exciting when the objects are active and dynamic than when they are passive or purely static. The state machine pattern is employed to provide active objects, or agents, in the virtual world that undertake predefined actions, but also respond to things that happen in the world as well.

This pattern is also useful for agents that may be user controlled in a multi-user context, but that need fall-back behaviour for situations where there are not sufficient users to control them. This way the virtual reality application can still provide a reasonable experience to smaller numbers of users.

The pattern for an autonomous agent assumes that we desire certain facilities in our agent:

- The agent moves around the virtual world, progressing from one location to the next in a sequence governed by its internal state. This is generalized so that the agent performs particular actions according to its state.
- The agent senses the world, and change its state in response to this, potentially affecting its movements.

The observant reader will notice that we build state machines for each agent in our virtual world. This potentially involves extra work because we might have to create different state machines for different types of object, and because we have to consider how each agent responds to events that affect it - events potentially caused by other agents. It would be tempting to just create one very large state machine that explicitly drives all of the agents in the virtual world¹. There is a virtual reality design philosophy behind embracing the per-agent approach. A large monolithic system is complex to build, but is also designed to follow only the

¹ Calvin and Hobbes, August 18 1986.

designer's expectation of how the VR application should behave. Individual response agents react to the conditions around them including to potentially unexpected situations. Interacting with live human users generates the unexpected situations. The combined responses from individual agents often results in emergent behaviour which may not have been explicitly defined in the application design, but that ensures that the application responds, stimulates and intrigues the participants in ways beyond its original design parameters.

The design for the agent specifies a set of states that the agent transitions through. While the agent is in one of these states it is doing activities related to that state. Thus it helps the design process if the name or description of the state defines the actions involved. Each state should also involve only a single action (ideally) otherwise the activity should be split into several states.

Events need to be defined for each state. Unless an action is intended to loop, there needs to be an event that leads to a subsequent state. This may be location related, for example, if the agent needs to register that it has arrived at a destination that it was moving towards. Destination locations can also be protected by invisible collision boxes and so the arrival could be noted through a collision event. External events that the agent is sensitive to need to be included in the state transition diagram. It is a worthwhile exercise to consider the effect of the external event with respect to every state in the diagram. This exploits this form of agent design to produce a robust agent able to respond appropriately under all scenarios.

6.2.2 *Pattern*

This pattern is a specific example of the state machine pattern presented in section 6.1 on page 205. This pattern elaborates on some of the details of processes specific to autonomous agents.

Consider an agent and the fragment of their state

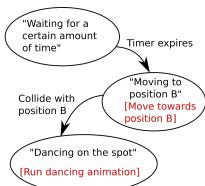


Figure 6.2.1: A portion of the state transition diagram wherein agents undertake specific operations while in a particular state.

transition diagram (Figure 6.2.1) that includes states:

Waiting for a certain amount of time: Event “timer expires” leads to state “moving to position B”

Moving to position B: Event “reached position B” leads to state “dancing on the spot”.

Dancing on the spot: Agent performs a dance action at its current position.

There are some benefits with not making too many assumptions about prior and final values of internal state values for the agents (such as position) when defining the states. Note the state involving movement to position B doesn’t make assumptions about the starting position (i.e. not a “move from A to B” state) and instead moves from wherever it is currently. This allows this state to be reused through events connecting other states to it.

Since there are time based events involved, we need to have a timer event handler that manages change of state:

```

function timerEventHandler (event ev)
{
    switch (state)

        case Waiting for a certain amount of time:

            changeState (moving to position B);
            break;

        default: // do nothing

    }
  
```

The timer event occurring in other states do not have any effect. Additional cases can be added to this event handler to support these cases if there are any other timer event emanating from other state in the state transition diagram.

We could use a similar event collision handler to detect when position B is reached. Assume that position B

is surrounded by an invisible collision boundary object named boundaryB.

```
function collisionEventHandler (event ev)
{
    switch (state)
        case Moving to position B:
            if ev.other == boundaryB
            {
                changeState (dancing on the spot);
            }
            break;
        default: // no nothing.
    }
}
```

The collision event provides details of the object, including a field named other which identifies the other object that this agent has collided with.

Collision events emanating from other states could be included by extending the switch statement.

As an alternative, reaching a particular location can be detected from the movement process. The actions which take place during the time period that the agent is in a particular state is discretely managed in the frame event handler function.

```
function update (deltaTime)
{
    switch (state)
        case Waiting for a certain amount of time:
            // no action required.
            break;
        case Moving to position B:
```

```

        Vector v = B - position;
        position = position + normalized (v) *
                    min (v.length, deltaTime)
        // Alternative check for reaching B
        if (v.length < distanceThreshold)
        {
            changeState (dancing on the spot);
        }
        break;

    case Dancing on the spot:

        ensure dancing animation is running
        break;

    default:

        // any states not listed that involve
        // no actions.
        break;
    }
}

```

The process of moving to position B from the current position involves finding a vector from the current position to B, and adding a scaled version of this to the current position. The scaling depends only on the timestep and not on the remaining distance to B to avoid Zeno's paradox. The length of the vector represents the distance from the current position to B, and we assume that we have reached B if this drops below a minimum distance threshold.

We could have also checked to see that the remaining distance to B was exactly zero. This is generally a bad idea since finite precision number representations and rounding errors are common in virtual reality contexts and exact equality of two quantities are rare. For the same reason, we use a minimum operator during moving to ensure that the last step arrives at the destination. Otherwise it would overshoot and spend an indefinite period

stepping backwards and forwards over the destination point.

Most virtual reality engines have facilities to play pre-recorded animations or motion captured actions which need to be triggered when entering a state and terminated when exiting a state. Functions to run once off code as a state is entered, or when it is exited can use the event produced by the changeState function.

```
function stateChangedEventHandler (oldState,
                                    newState)
{
    // handle leaving a state.
    switch (oldState)
        case Dancing on the spot:
            stop animation;
            break
        default: break

    // handle entering a state.
    switch (newState)
        case Dancing on the spot:
            start animation;
            break
        default: break
}
```

6.2.3 Example

Example 30. Applying the pattern to Unity software

1. This example uses the state machine pattern to specifically define the behaviours of an autonomous element. We model the behaviour of the fictitious Weeping Angel² from the Doctor Who series. Figure 6.2.2 shows the results of planning the actions of our planned agent in the form of a state transition diagram. After initialization the agent starts off in a Follow state, where they move towards the user's avatar.

² https://en.wikipedia.org/wiki/Weeping_Angel

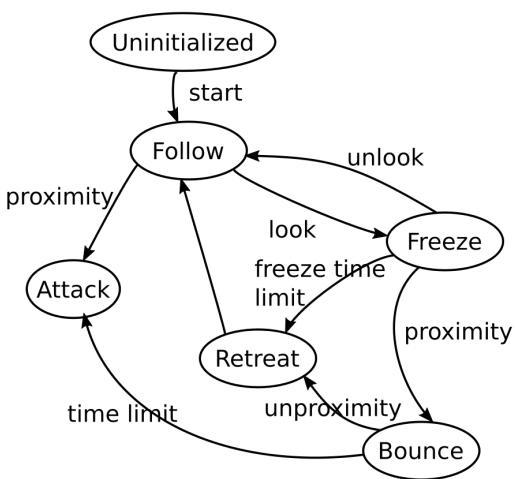


Figure 6.2.2: State transition diagram for the autonomous weeping angel agent.

If you look at them, they transition to the Freeze state, where they resume the actions expected of statues.

After a short period of time in this state however, they get nervous and start to retreat. Once you stop looking at them, they resume the Follow related actions. An agent in the Follow state that gets within a threshold distance of the user (a proximity event) enters the terminal attack state. Currently attacking involves turning red, and freezing, but other behaviour could be inserted for this. A user that moves closer to a frozen angel triggers its warning Bounce state. If they stay there too long the angel attacks them, but if they move out of range then the angel also starts retreating.

2. The virtual reality application starts with a new Unity software project. A rudimentary ground plane is provided in the form of a plane object.

The user's avatar is created with an empty object, which we make the parent of the main camera, in order to achieve the first person view. The camera may need to be positioned just above the origin relative

Algorithm 49 User controlled object steering component using the Unity software input system.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class SteerObject : MonoBehaviour {
6
7      public float speed = 0.1f;
8      public float turnspeed = 1.0f;
9
10     // Use this for initialization
11     void Start () {
12
13     }
14
15     // Update is called once per frame
16     void Update () {
17         float h = Input.GetAxis ("Horizontal");
18         float v = Input.GetAxis ("Vertical");
19
20         this.transform.position += speed * Time.deltaTime
21             * v * this.transform.forward;
22         this.transform.rotation *= Quaternion.AngleAxis (
23             turnspeed * Time.deltaTime * h, this.
24             transform.up);
25     }
26 }
```

to the parent object to achieve this. A C# SteerObject script is attached to the user's avatar. This script, in Algorithm 49, uses the Unity software input system to map horizontal and vertical axis controls (keyboard or joystick) into movement controls that are used to drive the avatar.

3. The scene itself is populated procedurally with prefab objects to represent each of the angels. The angel pre-fabs are created by constructing: a cylinder object with a new transparent material (the colour of this material is used to show the angel's state), and importing a child mesh holding the actual polygon mesh of the angel object. An externally sourced mesh was imported for this purpose. Attach a new C# script component

Algorithm 50 Yet another variation of the “add n objects in random positions” script.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class AngelGenerate : MonoBehaviour {
6
7      public int number0fAngels = 5;
8
9      public float radius = 10.0f;
10
11     public GameObject angelTemplate;
12
13     // Use this for initialization
14     void Start () {
15         for (int i = 0; i < number0fAngels; i++) {
16             Instantiate (angelTemplate, new Vector3 (Random.
17                         Range (-radius, radius), 0, Random.Range (-
18                         radius, radius)), Quaternion.identity);
19     }
  
```

called AngelBehaviour to this, before dragging the whole assembly into the Prefabs folder of your project’s assets.

The scene generator itself is a new empty object with a C# script component called AngelGenerate (Algorithm 50) attached. The previously created angel prefab must be provided as a parameter to this component in the Inspector pane.

4. The behaviour for an individual angel involves translating the state transition diagram into a script using the pattern. Each event handler (or function that is called when an event occurs) considers which state it is in, and acts accordingly. The intention with this example is to present a large enough range of states and events that the pattern becomes clear. Extending the behaviour of the angel is achieved by modifying or adding to the state transition diagram (Figure 6.2.2) and then translating it to the script by extending the

pattern already present.

For completeness, the entire script is included throughout the various pieces shown throughout the remainder of this example. As per the state machine pattern, the definition of the states, and the standard setState function are shown in Algorithm 51. A number of the particular constants used in the agent simulation are also named by being associated with variables of a particular name. This helps with any fine tuning of the process that may be required.

5. The core of the pattern is the handler for any particular event. This contains a switch statement, which ensures the response to the event is specific to the current state of the agent. Usually this involves an update to change to the state resulting as a consequence of the event occurring. Note that when the event has no affect in a particular state then no case is required in the switch statement for that state. The event handler functions for all of the events used in the state transition diagram for the weeping angel are listed in Algorithms 52 on page 231, 53 on page 232 and 54 on page 233.
6. Since some of these events are specific to the particular application (such as things changing when you look at them), we can generate these events ourselves. In many cases, this just involves using the Update function to test for a particular condition, and calling the event handler function directly if the condition is true. Algorithm 55 on page 234 and 56 on page 235 demonstrates how the Update function tests for conditions and calls event handler functions. It also contains an internal event handler, since the Update function is the event handler for the “nextFrame” event.
7. The remainder of the script consists of utility functions to check for visibility, and to generate timer events. These are shown in Algorithm 57 on page 236.

Algorithm 51 Standard state machine pattern elements for the weeping angel.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class AngelBehaviour : MonoBehaviour {
6
7      enum AngelStates { NotInit, Follow, Freeze, Retreat,
8                         Bounce, Attack };
9
10     // Current state in the state transition graph.
11     private AngelStates state = AngelStates.NotInit;
12
13     // speed of movement while following.
14     private float speedFollow = 0.5f;
15
16     // proximity threshold
17     private float proximityLimit = 3.0f;
18
19     // time limit on being frozen.
20     private float freezeTimeLimit = 2.0f;
21
22     // time limit on bouncing before attack.
23     private float bounceTimeLimit = 5.0f;
24
25     // the player that is being chased.
26     private GameObject player;
27
28     // coroutine used for the timer, so we can abort it
29     // if required.
30     IEnumerator timerCoroutine;
31
32     // This function updates the lawyer simulation
33     // according to the state involved.
34     void setState (AngelStates newState)
35     {
36         if (state == newState)
37         {
38             return; // nothing changed.
39         }
40
41         AngelStates oldState = state;
42         state = newState;
43         handleStateChangedEvent (oldState, newState);
44     }
```

Algorithm 52 Event handlers for the change state event.

```
1 // Event handlers. Each event handler should be well
   // behaved, and not take
2 // any longer than absolutely essential.
3
4 // Make any changes to the scene corresponding to
   // arriving in the given state.
5 // The previous state is provided, but usually not
   // required.
6 void handleStateChangedEvent (AngelStates oldState,
   AngelStates state)
7 {
8     // tidy up after any old states.
9     switch (oldState) {
10        case AngelStates.Freeze: clearTimer (); break;
11        case AngelStates.Bounce: clearTimer (); transform.
12            position = new Vector3 (transform.position.x,
13            0.0f, transform.position.z); break;
14    }
15
16    // For most new states, we set the colour of the
      // object, mostly to
17    // provide a visual indication of the current
      // state.
18    switch (state)
19    {
20        case AngelStates.Follow: this.GetComponent <
21            MeshRenderer> ().material.color = new Color
22            (1, 1, 1, 0.2f); break;
23        case AngelStates.Attack: this.GetComponent <
24            MeshRenderer> ().material.color = new Color
25            (1, 0, 0, 0.2f); break;
26        case AngelStates.Freeze: this.GetComponent <
27            MeshRenderer> ().material.color = new Color
28            (0.1f, 0.1f, 0.1f, 0.2f); startTimer (
29            freezeTimeLimit); break;
30        case AngelStates.Retreat: this.GetComponent <
31            MeshRenderer> ().material.color = new Color
32            (0.3f, 0.3f, 0.7f, 0.2f); break;
33        case AngelStates.Bounce: this.GetComponent <
34            MeshRenderer> ().material.color = new Color
35            (0.3f, 0.8f, 0.2f, 0.2f); startTimer (
36            bounceTimeLimit); break;
37    }
38 }
```

Algorithm 53 Event handlers for the collision event, and the proximity (within a threshold distance) events.

```
1 // All event changes resulting from collisions. This
   is the handleCollisionEvent function.
2 void OnTriggerEnter (Collider other)
3 {
4 }
5
6 // State changes as a result of getting too close.
7 void handleProximityEvent ()
8 {
9     switch (state)
10    {
11        case AngelStates.Follow: setState (AngelStates.
           Attack); break;
12        case AngelStates.Freeze: setState (AngelStates.
           Bounce); break;
13    }
14 }
15
16 // State changes as a result of being too far apart.
17 void handleUnproximityEvent ()
18 {
19     switch (state)
20    {
21        case AngelStates.Bounce: setState (AngelStates.
           Retreat); break;
22    }
23 }
```

Algorithm 54 Event handlers for the looked at (visible) events, and the timer event.

```
1  // State changes as a result of looking at a object.
2  void handleLookedAtEvent ()
3  {
4      switch (state)
5      {
6          case AngelStates.Follow: setState (AngelStates.
7              Freeze); break;
8      }
9
10 // State changes as a result of not looking at an
11 // object.
12 void handleUnlookedAtEvent ()
13 {
14     switch (state)
15     {
16         case AngelStates.Retreat: setState (AngelStates.
17             Follow); break;
18     }
19 // Respond to a timer event, based on what state we'
20 // re in.
21 void handleTimerEvent ()
22 {
23     switch (state)
24     {
25         case AngelStates.Freeze: setState (AngelStates.
26             Retreat); break;
27         case AngelStates.Bounce: setState (AngelStates.
28             Attack); break;
29     }
30 }
```

Algorithm 55 Testing for conditions and generating events if they occur (part 1).

```
1  void Start () {
2      player = GameObject.Find ("Avatar");
3      setState (AngelStates.Follow);
4  }
5
6  // Update is called once per frame
7  void Update () {
8      // generate any events required.
9      // Proximity.
10     Vector3 dir = player.transform.position -
11         transform.position;
12     if (dir.magnitude < proximityLimit) {
13         handleProximityEvent ();
14     } else if (dir.magnitude > 1.5f * proximityLimit)
15     {
16         // actually have to move a significant distance
17         // beyond proximity limit, to avoid change
18         // being too sensitive to small movements.
19         handleUnproximityEvent ();
20     }
21
22     // Visibility.
23     if (isVisible (40.0f)) {
24         handleLookedAtEvent ();
25     } else {
26         handleUnlookedAtEvent ();
27     }
28
29     // handle any updates specific to the current
30     // state.
31     switch (state)
32     {
33     case AngelStates.Follow:
34     {
35         dir = player.transform.position - transform.
36             position;
37         dir.Normalize ();
38         transform.position += speedFollow * Time.
39             deltaTime * dir;
40         transform.forward = dir;
41     }
42     break;
43     case AngelStates.Freeze:
44         // continued ...
45 }
```

Algorithm 56 Testing for conditions and generating events if they occur (part 2).

```
1      // continuation ...
2  case AngelStates.Freeze:
3      break;
4  case AngelStates.Retreat:
5  {
6      dir = -(player.transform.position - transform.
7          position);
8      dir.Normalize ();
9      transform.position += speedFollow * Time.
10         deltaTime * dir;
11      transform.forward = dir;
12  }
13  break;
14 case AngelStates.Bounce:
15 {
16     transform.position = new Vector3 (transform.
17         position.x, Mathf.Abs (Mathf.Sin (Time.
18         time)), transform.position.z);
19 }
```

Algorithm 57 Utility functions. Visibility tests are done by computing the (cosine of) angle between the user's forward direction, and the direction from user to the object containing this script component. If the angle is less than the provided threshold (or the cosine of the angle is greater than the cosine of the provided threshold) then the object is visible. Timer events use a coroutine. Care has to be taken to disable timers set when a particular state starts so they don't trigger in other states.

```

1   // Returns true if the current object is visible -
2   // within
3   // fieldOfView degrees of the player's direction of
4   // gaze.
5   bool isVisible (float fieldOfView)
6   {
7       Vector3 gazeDir = player.transform.forward;
8       // get the direction to the current object.
9       Vector3 actualDir = transform.position - player.
10          transform.position;
11       actualDir.Normalize ();
12
13       float cosAngle = Vector3.Dot (gazeDir, actualDir);
14       if (cosAngle > Mathf.Cos (fieldOfView * Mathf.PI /
15          180.0f)) {
16           return true;
17       }
18       return false;
19   }
20
21   // Sleep for a while and then generate a timer event
22   .
23
24   IEnumerator deferredTimerEvent (float timeDelay)
25   {
26       yield return new WaitForSeconds (timeDelay);
27       handleTimerEvent ();
28   }
29
30
31   // trigger a timer event after the specified time in
32   // seconds.
33   void startTimer (float duration)
34   {
35       timerCoroutine = deferredTimerEvent (duration);
36       StartCoroutine (timerCoroutine);
37   }
38
39
40   // clear any timers running at the moment.
41   void clearTimer ()
42   {
43       StopCoroutine (timerCoroutine);
44   }

```

8. The resulting virtual reality application produces effects such as shown in Figure 6.2.3. As the basis for a virtual reality application it requires quick reflexes to keep all angels away since they like to sneak up behind you. Be careful when getting close to frozen angels - they bounce for a while before attacking.

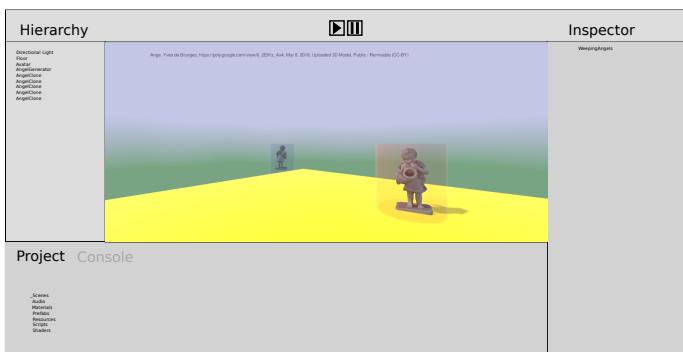


Figure 6.2.3: Weeping angels in action, showing angels in the attack (red) state, and others (blue) in the retreat state.

6.3 Variety and unpredictability

6.3.1 Description

The autonomous behaviours covered in the previous patterns have allowed complex behaviours which interact with, and respond to, other elements in the scene. This all contributes to rich virtual environment but can appear repetitive and predictable after a while. While not a replacement for artificial intelligence, a bit of randomness can create the illusion of more complex behaviour. Randomness is also a key ingredient in managing emergent systems and improving their robustness by ensuring that they can recover from failure. For example, an agent that directly follows the user may get trapped by some configurations of obstacles, but a bit of randomness in its movement may give it a chance to free itself.

Sources of randomness in software applications typically come from a random number generator. This is just a function that produces a long sequence of seem-

ingly very different values. This sequence is not actually random and the same sequence is generated the next time the application runs. We can get more variation by starting at a different point in the sequence each time we run the application. The starting offset, or seed, of the random number can be generated from a more convincing source of random numbers, such as the value on the microsecond counter of the computer's clock when the application is started.

Alternatively, if we want to reuse the same sequence of random numbers, we can force the seed to start at a predefined point each time. This is handy when looking for bugs in random agents, as it ensures they repeat the same sequence of actions and helps to identify both the cause of the problem, and ensures that you can check that the issue has been resolved.

Other functions are specifically designed to provide repeatable (deterministic) variety. The Perlin noise function (see section 9.2 on page 299) always provides the same output for the same input. If we use the current position of the agent as the input then we can expect it to behave the same way each time it visits that location. However behaviour in neighbouring locations may be very different.

The patterns below outline some of the ways randomness is used to provide variety to an agent; in its motion and in its behaviour.

6.3.2 *Pattern*

Random numbers are retrieved from the random number generator provided by the programming language and libraries used by the VR engine. Typical functions include:

```
r = randomFloat ()
```

Functions of this type create a floating point number; specifically any number in the range $[0, 1]$. You can extend this range by multiplying and shifting. For example:

```
r = 2 * randomFloat () - 1 // Range is [-1,1]
r = n * randomFloat () // Range is [0,n]
```

Some variations on this function do not include the upper bound i.e. $0 \leq r < 1$. The floating point random values are handy for choosing positions for placing objects or directions for moving them.

Integer versions of the random number function may also be available. These generally allow the range to be specified.

```
r = randomRange (l, u)
```

The value returned is a random integer in the range $[l, u)$. Note that it never achieves the upper value.

Some languages just provide a random function that returns any integer representable by computer. The modulus function (%) is used to map this back into a particular range:

```
r = l + (random () % (u - l))
```

Integer random numbers can also be achieved by rounding down scaled floating point random numbers. In this case, be aware of whether the upper bound is included in the range of the floating point numbers because actually achieving this value (and its corresponding integer) is very much less likely than achieving any of the other integer values in the range.

Generating random movement is a convenient way of providing some immediate activity in a newly created virtual world. The pattern below chooses a random direction to take a step at each frame:

```
function update (deltaTime)
{
    rx = 2 * randomFloat () - 1
    ry = 2 * randomFloat () - 1
    position = position + speed *
        Vector (rx, ry) * deltaTime
```

}

This movement resulting from this process resembles Brownian motion as each agent will continually be choosing a different direction every frame and following a random walk. A more deliberate but still random motion is achieved by choosing random way points, moving towards the first way point, and only adding further way points once the agent has reached the first one.

```
function update (deltaTime)
{
    if not at first way point
    {
        take one step along path towards way point
        face forward along tangent to the path
    }
    else
    {
        remove first way point from list
        rx, ry = random Position
        add (rx, ry) to way point list
    }
}
```

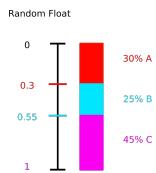


Figure 6.3.1: A given probability distribution is achieved by decomposing the range $[0, 1]$ of a typical random floating point value.

Simple motion might keep only one way point at a time and move directly towards that. The agent then changes direction sharply when it reaches that point and starts moving to the next. Several way points are used to define a spline path that passes through each way point and provide smoother motion.

The actions in the state machine can also be modified to include some random elements. For example, when a particular event X is triggered we can move to state A 30% of the time, to state B 25% of the time, and to state C the remaining 45% of the time. The code would like something like:

```

function eventXHandler (event ev)
{
    switch (state)
    ...
    case stateS:
        r = randomFloat ()
        if r <= 0.3 // 30%
        {
            changeState (StateA)
        }
        else if r <= 0.55 // 25% beyond the 30%
        {
            changeState (StateB)
        }
        else // remaining 45%
        {
            changeState (StateC)
        }
    ...
}

```

6.3.3 Example

Example 31. Applying the pattern to Unity software

1. This example focuses less on the mechanics of applying randomness in a virtual reality application but rather more on the implications of having random elements in a virtual world. We recreate a simulation known as a quincunx.³ This consists of a series of pegs on a vertical board. Objects dropped from the top bounce between the pegs and accumulate at the bottom. The shape of the pile at the bottom is intended to show that random events tend towards a normal (Gaussian) distribution as the bell shaped curve emerges in the shape of the pile of dropped objects.

³ also known as a Bean machine or Galton board
https://en.wikipedia.org/wiki/Bean_machine

This example creates a virtual reality version of this and allows the physics simulation of the virtual reality engine to control the behaviour of the dropped “beans”. This does illustrate the deterministic nature of computer simulations, even after reasonably complex sets of interactions. We then add some randomness to achieve a result that more closely matches expectations from physical reality.

2. The scene is created with a few of the usual ingredients. A large flattened cube is used as the base plate. The finite thickness of the cube is helpful with physical simulations as it reduces chances of a dropped object completely penetrating something thin (such as a plane) between two frames. Prefabs are also created for the pegs, and the beans. The pegs we used are spheres, since this ensures equal chances of bouncing evenly in any direction. The peg must have an active collider component. The bean uses a capsule, partly because the rounded edges also ensure even bouncing, but with a bit of asymmetry that could potentially introduce chaotic behaviour as it tumbles. The bean has a rigidbody component added to it, with gravity enabled.
3. The pegs could be placed individually by hand, but that would be tiresome. The script in Algorithm 58 on the facing page allows this to be managed procedurally, with the option of allowing experimentation with different configurations of pegs. An amusing aspect of the simulation results from having all the pieces confined to the same 2D plane (all the z coordinates used are 0). As a result there are no forces that push pieces in the z direction, so they stack nicely. At some point though, numerical instability pushes one piece slightly out of alignment and the entire stack collapses. The quincunx is sandwiched between two transparent cubes to help confine the fallen beans and help them stack up. A similar process could be used to create

Algorithm 58 Quincunx generation through placement of pegs in an offset triangular pattern.

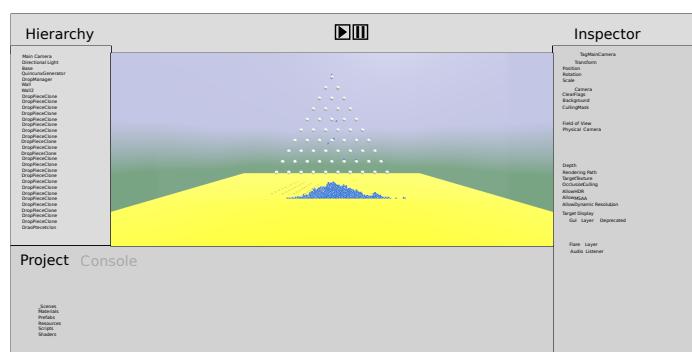
```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5 // Place pegs in the arrangement of a bean machine/
   Galton Board/quincunx
6 public class QuincunxGenerator : MonoBehaviour {
7     [Tooltip ("Position of the apex peg in the
               triangular formation")]
8     public Vector3 topPegPosition = new Vector3 (0, 20,
          0);
9     [Tooltip ("Horizontal gap between pegs")]
10    public float horizontalgap = 2.0f;
11    [Tooltip ("Vertical gap between pegs")]
12    public float verticalgap = 3.0f;
13    [Tooltip ("Number of rows of pegs")]
14    public int numberOfRows = 10;
15    [Tooltip ("The template for one peg")]
16    public GameObject pegTemplate;
17    // Place pegs in a staggered triangular formation,
       from the top point, with spacing defined by
       hgap horizontally, and vgap vertically. X axis
       is horizontal, Y axis is vertical.
18    void createQuincunx (Vector3 top, float hgap, float
          vgap, int numRows, GameObject peg)
19    {
20        int numElementsIn.CurrentRow = 1;
21        float startingX = top.x;
22        // lay down one row at a time.
23        for (int i = 0; i < numRows; i++) {
24            // lay down a row.
25            for (int j = 0; j < numElementsIn.CurrentRow; j
               ++
) {
26                GameObject pegPiece = Instantiate (peg, new
                   Vector3 (startingX + j * hgap, top.y - i
                     * vgap, 0.0f), Quaternion.identity);
27                pegPiece.transform.SetParent (this.transform);
28            }
29            // shift left edge.
30            startingX -= (hgap / 2.0f);
31            numElementsIn.CurrentRow += 1;
32        }
33    }
34    void Start () {
35        createQuincunx (topPegPosition, horizontalgap,
           verticalgap, numberOfRows, pegTemplate);
36    }
37 }
```

bins below the pegs, so bean piles cannot spread out.

4. Bean dropping involves creating new beans above the top of the quincunx and dropping them at regular intervals. Rate of dropping must be controlled so that beans don't collide with each other on the way down (although we do allow beans to dislodge any that get stuck on pegs). The script in Algorithm 59 on the next page provides the bean dropper functionality.
5. The version of the bean dropper in Algorithm 59 on the facing page has the option to include a random perturbation in the horizontal position from which the object is dropped. This is currently disabled (multiplied by 0) which means that each object is dropped from exactly the same position. The semi-chaotic nature of the physics simulation does still result in some variation in the final landing position for the beans, but this is quite restricted. You may observe that the first two beans dropped land perfectly balanced on the top peg, before the third one is enough to disturb the balance and send each of the three to a different pile. Similar but less likely interactions with some of the lower pegs produce some smaller piles of beans as well, as shown in Figure 6.3.2. In physical

Figure 6.3.2: In a deterministic simulation the bean distribution is less even, despite the collapse of piles producing some smoothing effects. The result is asymmetric, and uneven.



reality we would expect to see a more even distribution, or even one approximating the bell shaped Gaussian curve.

Algorithm 59 Bean dropper. This requires a change to the code to enable a random horizontal offset in the drop position.

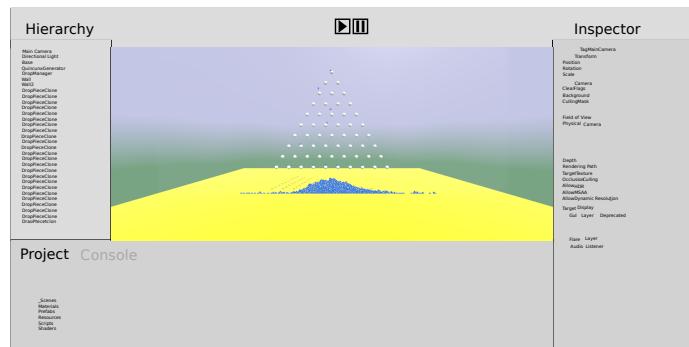
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  // Drop an object at regular intervals.
6  public class RegularRelease : MonoBehaviour {
7      [Tooltip ("Prefab of object for dropping")]
8      public GameObject dropObjectTemplate;
9
10     [Tooltip ("Time intervals in seconds between
11         dropping objects")]
12     public float timeInterval = 2.0f;
13
14     [Tooltip ("Position to drop the object from")]
15     public Vector3 dropPosition = new Vector3 (0, 200,
16         0);
17
18     // count time until next object ready for dropping.
19     private float counter;
20
21     // Use this for initialization
22     void Start () {
23         counter = 0.0f;
24     }
25
26     // Update is called once per frame
27     void Update () {
28         counter += Time.deltaTime;
29
30         if (counter > timeInterval) {
31             // drop object, potentially with random offset.
32             Instantiate (dropObjectTemplate, new Vector3 (0 *
33                 Random.Range (-0.01f, 0.01f),0,0) +
34                 dropPosition, Quaternion.identity);
35             counter = 0.0f;
36         }
37     }
38 }
```

6. We can enable the random perturbation in starting position by modifying the line of code as follows:

```
1 Instantiate (dropObjectTemplate, new Vector3 (1 *
    Random.Range (-0.01f, 0.01f),0,0) +
    dropPosition, Quaternion.identity);
```

This could represent the expected variation in positioning of subsequent beans that would happen in physical reality. The resulting distribution in the virtual reality simulation now more closely resembles that found in physical reality as shown in Figure 6.3.3.

Figure 6.3.3: Distribution of beans when a random perturbation is added to the system. The edges do also get emphasized as a result of beans getting extra energy when they collide which knocks them away from the triangular peg region.



7

Physical in Virtual

7.1 Collision detection

7.1.1 Description

Collision detection has many uses in virtual reality applications. A convincing sense of physical presence is supported by not allowing objects to interpenetrate. Interactions such as picking up objects or pressing buttons may be triggered by collisions between controllers and interface objects. Some locations even have invisible collision regions around them to detect when objects enter particular regions of the virtual world.

There are choices that need to be made regarding the nature of collision detection for any particular application. Each choice involves a trade off between the quality and accuracy of the collision detection process, and the time that it takes to perform it.

Precision: Identifying the exact point of collision between any pair of 3D objects is typically time consuming. Even where these objects are approximated with polygonal boundary representations, this process involves checking every polygon in one object against every polygon in the other object; a significant amount of computation. Efficiency is improved by grouping neighbouring polygons and approximating their shape with a simpler bounding shape (such as a box

or sphere). Most objects in virtual reality applications use such bounding volumes around each rigid component as the basis for collision detection.

Path: Virtual reality engines check for collisions at each frame (discrete collision detection). It is possible to miss collisions if objects move fast enough. For example, two fast moving objects may move completely through one another in the interval between two successive frames. This is addressed by building a bounding volume that represents the region of space traced out by the object during its entire motion from one frame to the next, and performing collision detection with these (continuous collision detection). The extra effort does degrade performance though.

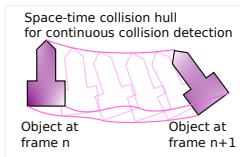


Figure 7.1.1: Continuous collision detection requires generating a bounding hull representing the region of space-time traced out throughout the interval between two frames.

Response: Typically *collision detection* refers to the process of determining *if* two (or more) objects are colliding. Determining what to do as a result of a collision is a process described as *collision response*. Having the collision response provided by the engine is convenient in some cases, such as preventing objects falling through the ground plane by making them bounce. In other cases it needs to be disabled, such as when picking up objects with the controller. Having a collision response which moves the object away each time the controller touches it would be awkward.

Collision detection involves selecting the appropriate configuration required. Choice is based selecting from the alternatives described above provided these are supported by the virtual reality engine. A collision event handler is then defined by each object involved in a collision in order to specify any further program actions that need to be taken in response to that collision.

7.1.2 Pattern

Some common configurations are suggested by each of the following patterns:

- Objects that are controlled by the physical simulation of the virtual reality engine, and which need to avoid moving through one another:

Ensure that each object has an appropriate bounding volume defined. Boxes and spheres can be efficiently checked for collisions. The bounding volume should be the best fit for the object shape. Some engines may support other bounding volume shapes which are a better fit in some cases. Planes can often have infinitely thin bounding volumes which might be replaced with a flattened box shape for a more robust result.

Discrete collision detection is better where many objects are involved, and they are not moving too rapidly.

Enable the engine's collision response to benefit from the physical simulation. Objects then bounce off one another according to their mass and elasticity properties.

- Detecting when a controller passes through another object.

The bounding volume for the controller can be reduced, if desired, to enclose only the active region of the controller. For example, if the controller is being used as a virtual wand with only the tip active, then the bounding volume would be a small sphere surrounding the tip of the wand object. A bounding volume is required for any object that the controller needs to interact with.

Discrete collision detection is usually sufficient, unless the controller needs to accurately respond when moved rapidly (e.g. sword-fighting) or intersect fast moving objects (e.g. ball-games such as tennis or cricket). If the engine allows, enable the continuous aspect only for the fast moving element to reduce the number of space-time hulls that need to be computed.

Disable the collision response. Often a collision de-

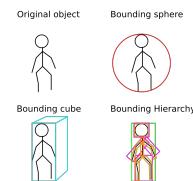


Figure 7.1.2: The efficiency of collision detection is improved at some cost to accuracy by enclosing the object in different formulations of bounding volume.

tection event handler would be added only to the controller object. This can then send controller specific types of events to the other object to make it respond in ways appropriate to the manipulation operation defined by the application's use of the controller.

7.1.3 *Example*

Example 32. Applying the pattern to Unity software

1. This example explores the process of collision detection. The scene used consists of a closed box. New objects are created at a constant rate, filling up the box. The behaviour of these objects as they collide can give insight into the properties of the virtual reality engine's collision detection and collision response systems.
2. Start with a new scene. We'll start with an enclosed box to contain the system of colliding objects. Use planes to build this box. Transparent materials on all but the bottom face help us view the objects inside. Planes are thin, so this configuration gives the most chance to demonstrate artefacts of the collision detection and response processes. Note that planes tend to be one sided, so make sure their front side is facing inwards into the box otherwise objects can roll through non-existent walls.
3. Create the prefab for the object that we use for demonstrating collisions. Start by adding an object such as a sphere to the scene. Make sure it has both a collider and a rigidbody component attached to it. Create a C# script component called CollisionBehaviour and also add this to the object. Then drag the object from the scene into the project window and place it in the Prefabs folder. The original can then be deleted from the scene.
4. We'll feed in objects to the scene slowly. Create a generator object (such as an empty) and attach a C#

Algorithm 6o A variation on the scene population pattern that adds objects at the same position but at different times.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class ObjectGenerator : MonoBehaviour {
6
7      // The object that will be created.
8      public GameObject objectPrefab;
9
10     // the number of objects created per second.
11     public float rate = 1.0f;
12
13     private float counter = 0.0f;
14
15     // Update is called once per frame
16     void Update () {
17         counter += Time.deltaTime;
18         if (counter > 1.0f / rate)
19         {
20             GameObject ob = Instantiate (objectPrefab);
21             ob.GetComponent <MeshRenderer> ().material.color
22                 = new Color (0.5f + Random.Range (0.0f,
23                         0.5f), 0.5f + Random.Range (0.0f, 0.5f),
24                         0.5f + Random.Range (0.0f, 0.5f));
25             counter = 0.0f;
26         }
27     }
28 }
```

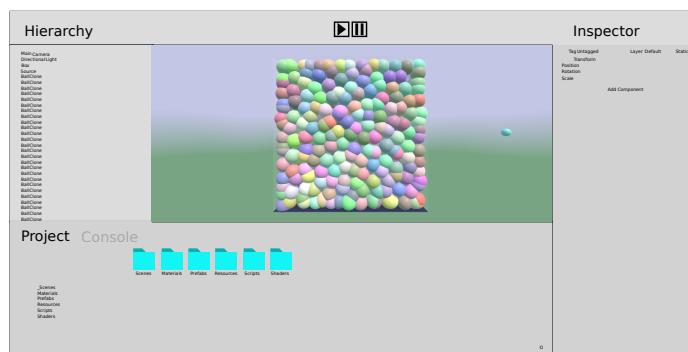
script called ObjectGenerator using the code contained in Algorithm 6o. Provide the prefab created previously as the value of the Object Prefab property.

5. The simulation at this stage is particularly revealing about the nature of the collision detection process, and about the physics simulation (section 7.3). Initially 1 object is created at a fixed point within the cube. The second is created at exactly the same location, a collision is detected and the default collision response forces the objects apart. Initially they are stacked vertically until the pressure builds up once the stack hits the roof, and a numerical instability blows this

apart. Ironically then the objects are still restricted to a vertical 2D plane, before eventually being forced to spread out across the floor.

As the box fills the balls are compressed and the collision response places greater and greater forces on the individual objects. Eventually this overwhelms the capabilities of the numerical solution to accurately perform collision detection and objects start bursting through the walls of the box when the force is enough to cause them to “tunnel” through the thin collider on the planes making up the box walls, as shown in Figure 7.1.3.

Figure 7.1.3: Collision configuration starting to produce large forces that cause discrete collision detection to start to fail.



6. There are a few options that are worth exploring before we build our own response to collision events. The rigidbody component has an option to change the collision detection process from discrete to continuous. It is worth experimenting with this setting and judging whether this resolves the tunnelling problem, or whether this solution comes at the expense of other collision detection properties (e.g. speed of simulation). The frame rate is shown by selecting the “Stats” control at the top of the Game window.

The other option to explore on our colliding object is the Trigger field provided as part of the collider. Enabling the Trigger checkbox causes the object to just report collisions rather than to trigger the collision

response. The objects now fall through the bottom of the box because they no longer respond to colliding with it.

7. With the trigger checkbox enabled, we'll build our own collision response function (which just imparts a small upward velocity to the object). Add the following event handler to the CollisionBehaviour script.

```

1 void OnTriggerEnter (Collider other)
2 {
3     this.GetComponent <Rigidbody> ().velocity = new
        Vector3 (Random.Range (-2.3f, 2.3f), 3.1f,
        Random.Range (-2.3f, 2.3f));
4 }
```

Since the principle action involved is to move upwards, this collision response function avoids obstacles below, but is not particularly robust when dealing with collisions at other angles.

The OnTriggerEnter event handler is only invoked if the trigger checkbox is ticked.

8. With the trigger checkbox disabled, we can augment the existing collision response. In this case, the event handler is OnCollisionEnter. Try out the following implementation of this event handler.

```

1 void OnCollisionEnter (Collision collision)
2 {
3     this.GetComponent <Rigidbody> ().velocity += new
        Vector3 (Random.Range (-0.1f, -0.1f), 0,
        Random.Range (-0.1f, 0.1f));
4     this.transform.localScale = 0.99f * this.
        transform.localScale;
5 }
```

This version attempts to reduce the pressure build-up within the box by reducing the size of the object by 1% each time it collides. Even with this small shrinkage factor the box no longer gets close to filling up as shown in Figure 7.1.4. A small random perturbation in the velocity also enough to avoid the stacking behaviour that is exhibited in the original simulation.



Figure 7.1.4: A modified collision event handler can customize the behaviour in response to a collision.

7.2 Forward Kinematics and Dynamics

7.2.1 Description

Kinematics refers to motion where we don't explicitly consider forces acting on objects but focus purely on the change in position and orientation of the object as a result of its velocity and acceleration (and rotational equivalents: angular velocity and angular acceleration). Acceleration results from the net force applied to an object, and consideration of forces is the topic of the field of dynamics.

Kinematic simulations are applicable once forces have been resolved (so once the dynamics simulation has determined acceleration) but are also convenient for simpler motions or where frequent user input needs to override or directly control aspects of the motion.

Mathematically, position is found by integrating velocity over time, knowing the initial position. Similarly velocity is found by integrating acceleration over time, knowing the initial velocity. Where acceleration is constant, analytical formulations of the relationship between position, velocity and acceleration are found as equations of motions. Effectively these are expressions that are directly evaluated at any point in time, to define the position, velocity and acceleration at that time.

Virtual reality applications tend to have cases where

acceleration is not constant, and where it is not even easy to determine except at discrete time steps. Direct manipulation of position to enforce motion bounds also limits the value of equations of motion. Instead virtual reality engines determine the position and orientation of an object over time (its trajectory) through a numerical approximation.

Various numerical approximations techniques exist that trade off computational effort against accuracy [Press et al., 2007]. Variations on the simpler kinematic pattern may be implemented in various forms while developing elements of virtual reality applications. Dynamics solutions are better suited to implementation within the game engine itself.

The discussion above has referred to kinematics in terms of position, velocity and acceleration but there are other contexts that may occur in specific virtual reality applications. Anything that involves a quantity whose behaviour over time is described by a rate-of-change is potentially a candidate for a kinematics solution. For example, consider a colour value that might change over time by selecting values at regular increments from a colour gradient chart.

7.2.2 Pattern

Assume we have three values; a position p , a velocity v and an acceleration, a . The velocity represents the rate of change of p over time, and the acceleration represents the rate of change of v over time. The goal is to determine future values of p and v , assuming that acceleration $a(t)$ is an input, potentially varying over time.

Given the discrete nature of the simulation, values are updated at discrete intervals, typically once per frame.

A typical Euler step update uses the following pattern:

```
function update (deltaTime)
{
     $\Delta t = \text{deltaTime}$ 
```

```

    a = getCurrentAcceleration ()
    v = v + aΔt
    p = p + vΔt
}

```

7.2.3 Example

Example 33. Applying the pattern to Unity software

1. This example builds the core mechanic for our new hit virtual reality experience: Tetchy Avians. We need to be able to launch small round objects into the air, and see them follow a suitable ballistic curve to impact elsewhere.
2. The scene is created with a large coloured ground plane. The launch device is represented with a cylinder to represent the barrel of a launch cannon. A C# cannon control script allows the barrel to be raised, or lowered, and some rotation from side to side, as per Algorithm 61 on the next page. The algorithm demonstrates some variations on scripting elements applied previously. Instead of using an if statement to provide the conditional pattern to constrain the cannon's movement angles this code uses the min and max operations.

The orientation of the cannon is controlled by two Euler angles. This operation then becomes very sensitive to the order in which the side to side and elevation rotations are performed. You'll notice a very different result if you swap the order of the two Quaternions.

3. The projectiles are round, coloured objects converted into prefabs. Since we're managing our own kinetics, we don't need to have any collision or physics components attached. Start with a sphere, set a suitable material and attach our kinetic control script as a C# component called AvianFlu. Convert this to a prefab and provide it as the avianPrefab property of the cannon.

Algorithm 61 A pan and tilt cannon capable of creating projectiles and providing them with their initial velocity.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class CannonControl : MonoBehaviour {
6
7      // limit of side to side movement.
8      public float sideAngleRange = 30.0f;
9
10     // limit of elevation
11     public float minElevation = 5.0f;
12     public float maxElevation = 80.0f;
13
14     // control sensitivity factor.
15     public float sensitivity = 10.0f;
16
17     private float elevation = 0.0f;
18     private float side = 0.0f;
19
20     public GameObject avianPrefab;
21
22     void Update () {
23         float v = Input.GetAxis ("Vertical");
24         float h = Input.GetAxis ("Horizontal");
25
26         side = Mathf.Max (Mathf.Min (side + h *
27                         sensitivity, sideAngleRange), -sideAngleRange
28                         );
29         elevation = Mathf.Max (Mathf.Min (elevation + v *
30                         sensitivity, maxElevation), minElevation);
31         this.transform.rotation = Quaternion.AngleAxis (
32             side, new Vector3 (0,1,0)) * Quaternion.
33             AngleAxis (elevation, new Vector3 (1,0,0));
34
35         if (Input.GetAxis ("Fire1") > 0)
36         {
37             GameObject avian = Instantiate (avianPrefab,
38                 this.transform.position, this.transform.
39                 rotation);
40             avian.GetComponent <AvianFlu> ().velocity =
41                 100.0f * this.transform.up;
42         }
43     }
44 }
```

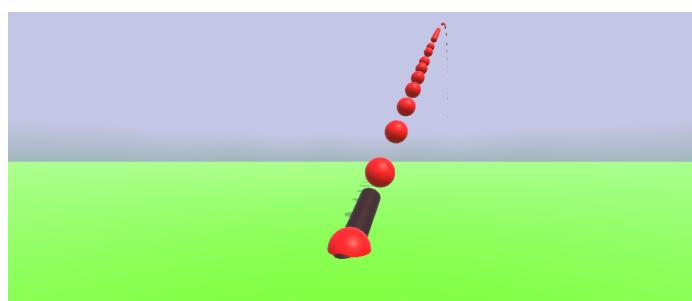
Algorithm 62 Forward kinematics for the avian projectiles.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class AvianFlu : MonoBehaviour {
6
7      private Vector3 acceleration = new Vector3 (0,-10,0)
8          ;
9
10     public Vector3 velocity = new Vector3 (0,0,0);
11
12     // Update is called once per frame
13     void Update () {
14         velocity = velocity + Time.deltaTime *
15             acceleration;
16         this.transform.position = this.transform.position
17             + Time.deltaTime * velocity;
18     }
19 }
```

The kinetic behaviour of the avian objects uses the forward kinematics pattern to perform numerical integration, converting acceleration to velocity, and velocity to position. The script for this is in Algorithm 62. The smoothing operations on the Unity software input system generate a sequence of non-zero values whenever the “Fire1” input is produced. This produces a stream of projectiles, following the expected parabolic ballistic trajectory, as shown in Figure 7.2.1.

Figure 7.2.1: The stream of projectiles launched from the avian cannon.



7.3 Inverse Kinematics

7.3.1 Description

In a scene containing a parent-child hierarchy (section 2.8 on page 63) such as a human armature, the kinematic and dynamics processes typically affect the root nodes, and pass on these effects through the parent-child links to each of the child nodes in turn.

In virtual reality applications we directly control some of the leaf nodes (those child nodes with no children of their own) in these hierarchies, specifically the head and the hands. To provide realistic behaviour for the avatar it is desirable to be able to calculate the effect of moving the hand on the intermediate and root nodes. Specifically, if the hand moves what happens to the lower arm, elbow, upper arm, shoulder and torso?

Inverse kinematics solves this problem.

The inverse kinematics problem is not one that has a single unique answer. There may be many configurations of shoulder and elbow position that all leave the hand in the same place. Inverse kinematics solvers try to use additional constraints to determine the most feasible solution (e.g. the ones that don't have the elbow bending the wrong way).

There are also cases where there is no valid solution. For example, most humans are unable to move their hands more than an arms length away from their bodies. In those cases, the inverse kinematics solver returns the best solution, such as placing the hand as close to the target point as possible, but at the limit of the arm's extension.

7.3.2 Pattern

Accessing the inverse kinematics functionality in the virtual reality engine involves:

1. Start with the parent-child hierarchy to which the inverse kinematics process must be applied. This may

be an armature, such as a human model.

2. Create a number of target objects, one for each leaf node that is controlled by inverse kinematics. These are invisible nodes, but they are used to designate the position and orientation that their corresponding body part must achieve.
3. Attach each body part to its target object. This is done by:
 - (a) indicating the weight (strength of the attraction) between target object and body part in the inverse kinematics properties.
 - (b) copy the position and orientation of the target object to the position and orientation properties of the body part.
4. Enable the inverse kinematics solver.

7.3.3 *Example*

Example 34. Applying the pattern to Unity software

1. The inverse kinematics facilities in Unity software are focused on specific properties of the humanoid armatures. The example provided in the Unity software documentation¹ demonstrates how the inverse kinematics pattern is applied in this engine. This requires a humanoid armature which may need to be retrieved from an appropriate source, such as the Unity store.

Start with a new Unity software scene. Add in a ground plane, and include the humanoid object. Make sure that it is a humanoid under the Rig tab of the import settings.

2. The humanoid object should have an Animator component if it has imported correctly. This needs an AnimationController asset, so create a new AnimationController in the project assets, and provide this as the Controller property of the Animator component.

¹ <https://docs.unity3d.com/Manual/InverseKinematics.html>

Under the Animator tab, select the Layers/Base Layer and use the gear (Settings) icon to tick the IK PAss checkbox.

3. The inverse kinematics solver can determine poses when the hands and feet are given target positions. The direction of gaze for the head can likewise be set. Create 5 small cubes to represent these targets. Create a C# script called MoveTargets using the code provided in Algorithm 63 on the next page, add the script to an empty object in the scene and provide these cubes as properties for this component. When running the application you should see each of the targets move, representing the desired position of each of the extremities. As yet, the humanoid does not move.
4. Use Algorithm 64 on page 263 to create a script called IKSolver, and add it to the humanoid object. Provide each of the target objects as properties. When run, the humanoid character will configure its joints to best place its extremities at the locations of the target objects, as shown in Figure 7.3.1.

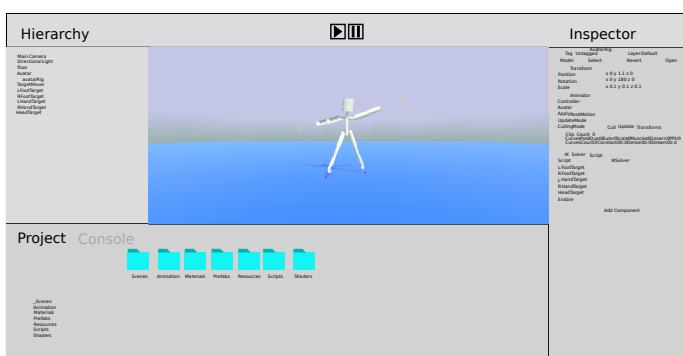


Figure 7.3.1: The inverse kinematics solver determines the body pose that positions the extremities at, or near, defined target points.

5. You can also disable the MoveTargets script, and manually move the target objects to get the range of motion possible through inverse kinematics. Target objects can be moved in scene view, or by directly altering position properties in the inspector window.

Algorithm 63 Each of the target points are moved using periodic functions.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MoveTargets : MonoBehaviour {
6      public GameObject LFootTarget;
7      public GameObject RFootTarget;
8      public GameObject LHandTarget;
9      public GameObject RHandTarget;
10     public GameObject HeadTarget;
11
12     public float speed = 1.0f;
13     public float stretch = 1.0f;
14
15     void Update () {
16         LFootTarget.transform.position = new Vector3 (0.5f
17             , 0, stretch * Mathf.Sin (speed * Time.time))
18             ;
19         RFootTarget.transform.position = new Vector3 (-0.5
20             f, 0, -stretch * Mathf.Sin (speed * Time.time
21             ));
22         LHandTarget.transform.position = new Vector3 (1.5f
23             , 1.0f + stretch * Mathf.Cos (speed * Time.
24             time), stretch * Mathf.Sin (speed * Time.time
25             ));
26         RHandTarget.transform.position = new Vector3 (-1.5
27             f, 1.0f + stretch * Mathf.Sin (speed * Time.
28             time), stretch * Mathf.Cos (speed * Time.time
29             ));
30         HeadTarget.transform.position = new Vector3 (
31             stretch * Mathf.Sin (speed * Time.time), 1.5f
32             + 0.5f * stretch * Mathf.Sin (speed * Time.
33             time), stretch * Mathf.Cos (speed * Time.time
34             ));
35     }
36 }
```

Algorithm 64 Inverse kinematics for any extremity involves setting the weight for that extremity to 1, and defining the position and orientation of the corresponding target object.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class IKSolver : MonoBehaviour {
6      public GameObject LFootTarget;
7      public GameObject RFootTarget;
8      public GameObject LHandTarget;
9      public GameObject RHandTarget;
10     public GameObject HeadTarget;
11     public bool enable = false;
12
13     void trackPart (AvatarIKGoal part, GameObject target
14         ) {
15         Animator a = GetComponent <Animator> ();
16         a.SetIKPositionWeight (part, 1);
17         a.SetIKRotationWeight (part, 1);
18         a.SetIKPosition (part, target.transform.position);
19         a.SetIKRotation (part, target.transform.rotation);
20     }
21
22     void untrackPart (AvatarIKGoal part) {
23         Animator a = GetComponent <Animator> ();
24         a.SetIKPositionWeight (part, 0);
25         a.SetIKRotationWeight (part, 0);
26     }
27
28     void OnAnimatorIK () {
29         if (enable) {
30             trackPart (AvatarIKGoal.LeftFoot, LFootTarget);
31             trackPart (AvatarIKGoal.RightFoot, RFootTarget);
32             trackPart (AvatarIKGoal.LeftHand, LHandTarget);
33             trackPart (AvatarIKGoal.RightHand, RHandTarget);
34             GetComponent <Animator> ().SetLookAtWeight (1);
35             GetComponent <Animator> ().SetLookAtPosition (
36                 HeadTarget.transform.position);
37         } else {
38             untrackPart (AvatarIKGoal.LeftFoot);
39             untrackPart (AvatarIKGoal.RightFoot);
40             untrackPart (AvatarIKGoal.LeftHand);
41             untrackPart (AvatarIKGoal.RightHand);
42             GetComponent <Animator> ().SetLookAtWeight (0);
43         }
44     }
45 }
```

8

All Together

8.1 Multiple users in multi-user virtual worlds

8.1.1 Description

Shared virtual worlds allow multiple users, each with their own equipment and at separate locations, to participate in the same virtual environment. Typical minimal expectations in these scenarios is that each user is able to see representations of the other users (their avatars), and that any user is able to interact with objects in the scene with the outcome of that interaction visible to all. Some mechanism for avoiding contention, when multiple users try to manipulate the same object at the same time, is required.

Some elements may be relevant only to individual users and not be shared. This might include elements of pop-up menus or other user interface support elements that are either private, or not pertinent to others.

Shared virtual worlds would be achieved by making use of the networking functionality supplied by the virtual reality engine, or any compatible networking libraries. There are a few key terms which the multi-user VR application designer should be aware of when deciding on the technologies to employ:

Client-server versus peer-to-peer: Most engines support a client-server model, where every user is considered a

client. The virtual world is stored and manipulated on a separate server device. Clients are only responsible for collecting input from the user, sending this information to the server, receiving details of the updated virtual world (as a result of the interaction of *all* clients), and presenting a view of the virtual world through the hardware attached to that client. The server receives details of the input from the client, modifies the representation of the virtual world (the information only, the graphical presentation occurs on the clients), performs any global tasks such as managing physics simulations, *resolves any contentions*, and send updates to every client. Each client maintains copies (clones) of the information (properties) of all objects in the world. Since these are only updated from the master copy on the server they are sometimes referred to as *imposters*.

The client-server model benefits from the simplicity of the structures and the centralized control of the system. However the server is a potential bottleneck as all communication needs to be routed through this device. Peer-to-peer networks share the work of the server between the devices of all participating users. For example, the volume of the virtual environment might be divided up into different regions, and each peer operates as a server for one of these regions.

More advanced peer-to-peer systems would duplicate this information and have fail-over strategies in the event of a peer leaving the network. Such architectures involve greater complexity in managing networking but scale better with load (the more peers join the network, the greater the capacity for managing the virtual world) and are more fault tolerant.

Several compromise arrangements are possible. The bottleneck of the server in the client-server architecture is reduced by spreading the load across several server devices. This replaces with server at the core of the network with a peer-to-peer cluster of server nodes, still at the core of the network. The behaviour of the

client remains unchanged, so this achieves some of the logical simplicity of the client-server architecture as well as improved redundancy and capacity of a peer-to-peer topology.

Small scale shared environments may elect to avoid having a dedicated server device and run the server on the same device as one of the clients. This is done by either separating them logically as separate services, or by allowing the server to include the client functionality of directly receiving input and presenting (graphically) the view of the virtual world. The latter configuration with combined client and server is a *host*.

Real-time versus reliable: Modern computer networks tend to support two different classes of service. Reliable connections between two devices (such as that provided by internet protocols such as TCP) ensure that the data received arrives in the same order, recovers any data lost during transmission, and at a rate consistent with the capacity of the network. This involves a significant amount of negotiation behind the scenes between the communicating devices and the application of extreme amounts of patience. A reliable connection typically waits for as long as it takes to ensure it provides the required reliable service. Only if communication fails completely will it return an error message.

Real-time communication (for example, as provided by internet protocols such as UDP) sends any message exactly once. This message is sent to multiple destinations at once (using multicast or broadcast addressing), but its arrival at any destination is subject to the whims of the network. It may be delayed by congestion on its path towards the destination, or even discarded without notification by overloaded routers. Messages may even arrive out of order, if a later message discovers a shorter path has become available.

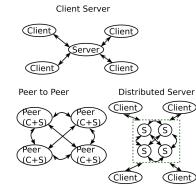


Figure 8.1.1: A combination of the centralized client-server model with a robust peer-to-peer server core allows the networked virtual reality system to scale, and reduces issues associated with failure of a single server.

Long messages need to be broken down into smaller packets to traverse the network and each of these fragments may have a different experience (e.g. some of them may not arrive at all). However when they do arrive, they arrive as soon as possible given current conditions on the network.

Multi-user virtual reality systems require both types of services. Reliable connections are required for sharing large amounts of data that changes infrequently. The initial state of the virtual world, and the geometric description of the objects involved, is distributed to new clients joining the world using this service. Real-time communication is used for updates to the virtual world. These updates, such as the current position of a moving object, represent transient information that is superseded by later updates to the same object. The effect of a missing or out-of-order packet produces at most a short-lived glitch in the virtual world, and even these can be avoided with a bit of extra diligence on the client's part (e.g. dead-reckoning position for missing packets, numbering the messages to detect out-of-order message).

The user in a multi-user virtual world is a special case object that has certain privileges. It occupies a special role on the local client, being the only object that should respond to input from the virtual reality hardware on that device. It also is the only object with the authority to modify the properties of that user. The server refuses updates from other clients that try to manipulate your avatar. In common with other shared objects, updates to the user's properties (such as position) are shared with other clients so that the imposter version of the user on other machines mirrors the actions that take place on the local device.

8.1.2 Pattern

Developing a multi-user virtual reality application requires the cooperation of the networking facilities in the virtual reality engine. This must be activated in a way specific to the particular engine being used, such as by adding a particular (invisible) object to the scene.

```
add network manager to scene
```

A particular object template for user avatars then needs to be registered with the engine, ensuring that the virtual world supports multiple users. When a new client joins the shared virtual world, the engine instantiates this template on the server to represent the user, replicates this object as an imposter to all of the clients, and associates that object with the client that has just joined. In this way, the object is able to check whether it is running on its local machine or not. Remember that every imposter is running the same script on all clients.

```
add the avatar template to the appropriate
field in the network manager
```

The user's avatar control script retrieves input from the hardware attached to the device and use this to manipulate its own properties - specifically updating position and orientation to move the avatar object. It must check that it is the local avatar object before doing this. Failure to do this may result in *all* avatar imposters responding to input on other machines and the server *will not allow this* to be updated consistently across all devices.

```
function update ()
{
    if (isLocalAvatar)
    {
        get controller input
        update position and orientation properties
    }
}
```

```
}
```

The updated position and orientation values need to be sent to the server to be distributed to other clients. A virtual reality engine often handles common property updates (such as position and orientation) by just tagging the object as requiring this facility. This might involve checking a checkbox in the network manager object, or inheriting the player script from a NetworkedObject or similar parent class. Updates to other custom properties would need to be done manually, as per the management of shared objects pattern in section 8.3 on page 281.

```
class AvatarScript implements NetworkedObject
{
    function update ()
    {
        ...
    }
}
```

8.1.3 Example

Example 35. Applying the pattern to Unity software

1. This example builds the core elements of a multi-player virtual reality racetrack, where players can race each other around a fixed course. Physics is enabled, allowing players to push each other around.
2. We start with the single player version, to ensure that the scene is appropriately set up. This involves including the following elements, as shown in Figure 8.1.2 on page 272:
 - (a) The race track. A large flattened cube can make up the base. Use stretched cubes to provide the barriers on the side of the road. You can make the track as complex as you like, but for simplicity we use a square.

Algorithm 65 Force based movement controls.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class SteerCar : MonoBehaviour {
6      void Update () {
7          float h = Input.GetAxis ("Horizontal");
8          float v = Input.GetAxis ("Vertical");
9
10         float forceEffect = 100.1f;
11         float angleRate = 10.0f;
12         GetComponent <Rigidbody> ().AddForce (forceEffect
13             * v * transform.forward);
14         transform.localRotation *= Quaternion.AngleAxis (
15             angleRate * h, transform.up);
16     }
17 }
```

- (b) A player prefab. This is the vehicle that the player drives. Import or assemble a suitable car shape, and convert this into a prefab.
- (c) A player control script. We use forces to control the vehicle, to ensure that we don't override any of the physics and collision effects. Add a rigidbody component to the player vehicle, and the script component called SteerCar shown in Algorithm 65. Depending on the shape of your vehicle, the forces applied can make the wheels dig in, causing the car to tumble. One strategy for addressing this is to add a physics material to the collider components of the ground and wheels. Reducing the friction settings in this physics material helps the vehicle move more easily. (The other option is to turn off gravity, but then you're creating a space ship simulator).
- You should be able to test this out by adding one or more of the vehicle prefabs to the scene, and ensure that you can drive around the race track with it.
3. The multi-player version of this application involves applying the manager pattern (section 2.9 on page 69)

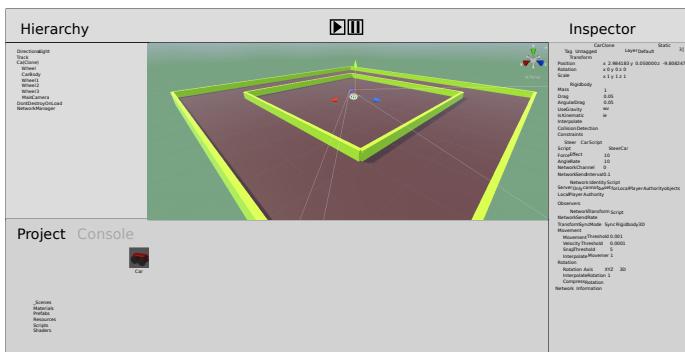


Figure 8.1.2: Example of race track layout, with one of the vehicle prefabs placed in the center.

using an existing Unity software component on the manager object. Create an empty object named NetworkManager and add a Network/NetworkManager component to this.

The vehicle prefab needs to be provided to this network manager component in the Player Prefab field. In order to do this, you need to first add a Network/NetworkIdentity component to the vehicle prefab.

We also need a mechanism to start a client or a server. Add a Network/NetworkManagerHUD component to the network manager object.

When running the virtual reality application, the HUD component allows three modes of operation:

- Host: application runs both a client and a server on the local machine.
- Server: only the server is run, so no players are created.
- Client: the address of the server machine is required, and the client must be able to connect to this before starting.

Using the host option confirms that the application runs, and that a new vehicle is created to represent the player on this client.

Testing multiple players is tricky and is best achieved using multiple machines. However to support rapid

development it is convenient to run a second copy of the application on the same machine, as shown in Figure 8.1.3. This is done by building and running the application (under File/Build and Run). The version running under the Unity software editor represents one player, the built application represents a second player. The built application can be run multiple times if more clients are required. Note that running multiple applications on a single machine does not actually generate any network traffic over your network connection (it is all simulated through an internal loopback virtual network device). Any issues with network performance and reliability are not noticed until you start testing on a physical network. Testing



Figure 8.1.3: Two networked players, one running in a built application and the other through the Unity software editor.

the application at this stage shows that each client adds a new vehicle to the virtual world. However both vehicles are controlled by the input on the local client, and no position updates are propagated to the other client.

4. We can ensure that player input only applies to their particular vehicle by using a flag variable provided by Unity software: `isLocalPlayer`. To access this variable the `SteerCar` class needs to inherit from `NetworkBehaviour`, rather than `MonoBehaviour`. We also need to import `UnityEngine.Networking` to access this class. The resulting code is shown in Algorithm 66. We still

Algorithm 66 Vehicle control that is network aware and applied only to the local player.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.Networking;
5
6  public class SteerCar : NetworkBehaviour {
7
8      public float forceEffect = 10.0f;
9      public float angleRate = 10.0f;
10
11     void Update () {
12         if (isLocalPlayer)
13         {
14             float h = Input.GetAxis ("Horizontal");
15             float v = Input.GetAxis ("Vertical");
16
17             GetComponent <Rigidbody> ().AddForce (
18                 forceEffect * v * transform.forward);
19             transform.localRotation *= Quaternion.AngleAxis
20                 (angleRate * h, transform.up);
21         }
22     }
23 }
```

don't have changes to the player being shown on the other player's view of the world.

5. Sharing properties of objects, such as position or colour, is covered in section 8.3 on page 281. However updates to the transform component's properties are so common that a dedicated component has been created to provide this. Add a Network/NetworkTransform component to the vehicle prefab. This ensures that updates to the position, rotation and scale properties are sent to the server, and then shared with the other clients.

You need to set the Local Player Authority property in the Network Identity component of the vehicle prefab. This ensures that the modifications to the player's object on their client are considered authoritative, and won't be overridden by changes (such as physics simulations) on other machines.

6. A final touch might be to associate the local client's camera with the local player's vehicle, so that each player can get a first person view on their machine. The code fragment below searches the scene for a camera with the name "Main Camera" (the Unity software default) and makes that a child of the current player.

```

1 void Start ()
2 {
3     if (isLocalPlayer)
4     {
5         GameObject camera = GameObject.Find ("Main
6             Camera");
7         if (camera != null)
8         {
9             camera.transform.parent = this.transform;
10            camera.transform.position = new Vector3 (0,
11                3, 0);
12        }
13    }
14 }
```

8.2 Network Spawn

8.2.1 Description

Adding an object to a multi-user virtual world is more complex than just adding it to a single user environment. Following in the conventions of some virtual reality engines, we *instantiate* objects in a single user context (section 2.3 on page 33), and *spawn* them in a multi-user context.

In the multi-user situation, creating shared objects is accomplished by sending a request to the server which then creates the object in its master representation of the world. Imposters for that object are then created on each of the clients, including the one which requested the object creation, as the server shares updates with each client.

8.2.2 Pattern

The typical pattern for spawning an object in a networked virtual reality is:

1. Register the object as a network shareable object.
This might involve adding it to a list managed by a network manager, or adding a particular component or interface to the object. This allows the network subsystem to ensure that any required properties are included in the object, such as identifiers to keep track of the type and specific instance of each object.
2. The client issues a request to the server to create a particular object. This is achieved through messages or remote procedure calls using the patterns in section 8.3 on page 281.
3. The server instantiates the object and ensures that copies are shared with all clients. This may require the object to be first instantiated on the server (section 2.3 on page 33) before being passed to a game engine specific spawn function.

8.2.3 Example

Example 36. Applying the pattern to Unity software

1. The network spawning pattern is demonstrated through creating a collaborative block building application. Each player is able to add blocks to the shared virtual world, and the resulting structure is the combined effort of all the participants.

Starting with a new scene, an Empty object named NetworkManager is added, and equipped with the NetworkManager and NetworkManagerHUD components.

A block prefab is created, consisting of a cube with a Rigidbody with gravity enabled, and represents one of the stackable blocks that are used.

A player prefab is created, consisting of a suitable shape (a thin cylinder is used in the images shown), and with a script named LaunchBlock to initially launch blocks, as shown in Algorithm 67. Ensure that the block prefab is provided as a property to this script component. A ground plane provides a base for objects to stack themselves.

2. The player prefab then needs a NetworkIdentity component added, after which it is registered by adding it to the Spawn Info/Player Prefab property of the NetworkManager. This provides the ability to run multiple copies of the application and have a player in each. However none of these share any of their state with the server or any other client.

As per the multi-user pattern, we still need to:

- (a) Set the script to inherit from NetworkBehaviour rather than MonoBehaviour (remembering to also add “using UnityEngine.Networking;”).
- (b) Ensure that response to controls is protected by an “if (isLocalPlayer)” conditional.

Algorithm 67 A turret capable of launching blocks.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class LaunchBlock : MonoBehaviour {
6
7      public GameObject blockPrefab;
8
9      public float turnSpeed = 10.0f;
10
11     public float launchForce = 5.0f;
12
13     public float fireInterval = 0.5f;
14
15     private float timeTillNextFire = 0.0f;
16
17     // Update is called once per frame
18     void Update () {
19         float hori = Input.GetAxis ("Horizontal");
20         float fire = Input.GetAxis ("Fire1");
21
22         transform.rotation *= Quaternion.AngleAxis (hori *
23                                         turnSpeed * Time.deltaTime, transform.up);
24         if ((timeTillNextFire < 0.0f) && (fire > 0.5f))
25         {
26             GameObject block = Instantiate (blockPrefab,
27                                             transform.position + transform.forward,
28                                             Quaternion.identity);
29             block.GetComponent <Rigidbody> ().AddForce (
30                 launchForce * (transform.forward +
31                                 transform.up), ForceMode.Impulse);
32             timeTillNextFire = fireInterval;
33         }
34         timeTillNextFire -= Time.deltaTime;
35     }
36 }
```

- (c) Adding a Network/NetworkTransform component to the player prefab to ensure that its position and orientation are shared with the server and other clients.
 - (d) Set the Local Player Authority property in the NetworkIdentity component of the player prefab to ensure that clients are allowed to update their data on the server.
3. At this point, all players exist and are updated in each of the different clients. However the blocks placed are local to each of the clients. These need to be spawned rather than instantiated so that they are created on the server, and then shared with all clients. The steps to achieve this include:
- (a) Add a NetworkIdentity component to the block prefab.
 - (b) Add the block prefab to the Registered Spawnable Prefabs list property of the NetworkManager.
 - (c) Add a NetworkTransform component to the block prefab, to ensure that changes in its position and orientation are shared.
 - (d) After instantiating the block in the script, use NetworkServer.Spawn to register it with the server. This function needs to be run on the server, so we separate it out into its own function that is marked with the Command attribute.
4. The final player script appears as shown in Algorithms 68 and 69. Figure 8.2.1 shows the application in action on the host device, after blocks have been added by different clients. Note that the colour of the blocks is not shared across different devices (only the position and orientation because of the use of the NetworkTransform component). Sharing custom properties involves use of a different pattern.

Algorithm 68 Use a network spawn operation to ensure that objects are created on the server and are replicated as imposters across all the clients (part 1).

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.Networking;
5
6  public class LaunchBlock : NetworkBehaviour {
7      public GameObject blockPrefab;
8      public float turnSpeed = 10.0f;
9      public float launchForce = 5.0f;
10     public float fireInterval = 0.5f;
11     private float timeTillNextFire = 0.0f;
12     // Give each player their own colour.
13     private Color blockColour;
14
15    void Start () {
16        blockColour = new Color (Random.Range (0.0f, 1.0f)
17                           , Random.Range (0.0f, 1.0f), Random.Range
18                           (0.0f, 1.0f));
19
19    [Command]
20    void CmdSpawn ()
21    {
22        GameObject block = Instantiate (blockPrefab,
23                                         transform.position + transform.forward,
24                                         Quaternion.identity);
23        block.GetComponent <Rigidbody> ().AddForce (
25            launchForce * (transform.forward + transform.
26            up), ForceMode.Impulse);
24        block.GetComponent <MeshRenderer> ().material.
25            color = blockColour;
25        NetworkServer.Spawn (block);
26    }
27    // continued ...

```

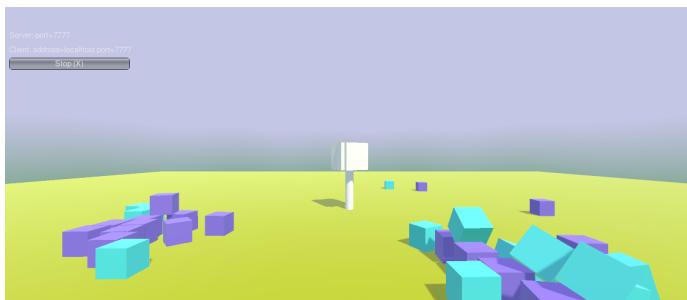


Figure 8.2.1: A multi-player block placement application.

Algorithm 69 Use a network spawn operation to ensure that objects are created on the server and are replicated as imposters across all the clients (part 2).

```

1  // continuation ...
2  // Update is called once per frame
3  void Update () {
4      if (isLocalPlayer)
5      {
6          float hori = Input.GetAxis ("Horizontal");
7          float fire = Input.GetAxis ("Fire1");
8
9          transform.rotation *= Quaternion.AngleAxis (hori
10             * turnSpeed * Time.deltaTime, transform.up
11             );
12
13          if ((timeTillNextFire < 0.0f) && (fire > 0.5f))
14          {
15              CmdSpawn ();
16              timeTillNextFire = fireInterval;
17          }
18      }
19 }
```

8.3 Shared objects in multi-user virtual worlds

8.3.1 Description

Changes to objects in a multi-user virtual world need to be immediately visible to all users on every client in the virtual world. These changes include updates to position and orientation, but also any other update to the object's properties that are pertinent to other users in the virtual world. In practice, changes are not immediate as they have to be passed onto the server and then relayed to all other clients.

The process of updating properties of shared objects now involves several steps:

1. A function (an event handler) is called for the object instructing it to update the value of one of its properties. This call takes place on one of the clients. The client must not directly modify the property for several

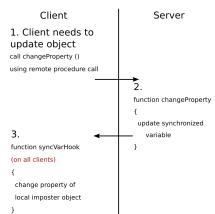


Figure 8.3.1: An update to a shared object on a client requires a modification to the data structure on the server, and then the server must replicate this change on every client.

reasons: this would immediately produce inconsistency with other clients, and we have not yet checked if the user on that client is even allowed to modify this object. Instead, the details of this change must be sent to the server, which decides if the request is permissible, and resolves contention if two users try to modify the same object at the same time.

2. If the server agrees to the change then it modifies the property on its own master copy of the scene graph. This ensures the change is available to new clients who may only join the shared world at some later point in time.
3. The server then issues an update to all clients, including the one who requested the change in the first place. When the client receives this instruction from the server it can go ahead and modify the property on the local imposter for the changed object.

There are several common mechanisms employed by networked virtual reality engines to achieve this communication:

Messaging: A data object consisting of just the information to be communicated is created and explicitly sent to the destination device. This needs to be converted into a sequence of bits to travel across the network; a process known as serialization. Serialization is a service supported by the VR engine, but limitations on the size of network packets and on the type of information that can be meaningfully reconstructed at the destination (references to other objects that exist only on one machine are difficult to manage) favour keeping these messages relatively simple and constructed from primitive types. Once the message is received, it is deserialized and passed to a network message event handler.

Remote procedure calls: This also performs the messaging process but disguises it as a call to a function. The

special trick is that the function appears to run on the destination machine. The parameters passed to the function become the contents of the message that is sent. The code within the function is the event handler that runs when the message is received.

Synchronized variables: This is another cosmetic modification to the messaging approach. Individual properties are designated as synchronized variables. When they are modified on the server, a message with the new value is sent to each client, and a particular event handler is run on each client allowing the client to respond, usually by updating the value of a local property on the client.

8.3.2 Pattern

The pattern below shows the process of updating property p of an object using a remote procedure call paradigm.

Let us assume that the object supports an interface (an externally visible function or event handler) that other objects or users on any client can call to update property p . This avoids modifying the object's property directly, but instead relays the change to the server.

```
function changeProperty (newValueForP)
{
    serverChangeProperty (newValueForP)
}
```

The function `serverChangeProperty` is another very similar looking function but this one runs on the server. It must be designated as being a server based remote procedure call in some way, potentially by using features of the programming language used to mark up particular functions.

[ServerRPC]

```

function serverChangeProperty (newValueForP)
{
    if (isServer)
    {
        syncP = newValueForP
    }
}

```

A basic precaution is included, to check that the server-ChangeProperty function is actually running on the server. This guards against accidental (or deliberate) use of the function by a badly written script on one of the clients. The server may also restrict which objects are allowed to use remote procedure calls. For example, one user is usually not able to call the remote procedure of another user.

The server is using a synchronized variable, syncP. When this value is updated, the update is sent to every client. The pattern for the synchronized variable is:

```

[SynchronizedVariable: eventHandler =
clientChangeProperty]
Type syncP

```

The annotations added to the declaration of the variable indicate that when it is updated on the server, a message is sent to each of the clients, and each client runs a function as defined by the value assigned to eventHandler.

The final step is to define the function which runs on each of the clients.

```

function clientChangeProperty (newValueForP)
{
    p = newValueForP
}

```

8.3.3 Example

Example 37. Applying the pattern to Unity software

1. The ability to modify custom properties and have these changes reflected on all the other imposters of the same object is demonstrated in this example by creating a shared notice board. Messages that are added via any one of the clients are then relayed to the server, and updated on all other copies of the notice board on the other clients.

The network infrastructure of the application is set up as follows:

- An empty object serves as the network manager, with NetworkManager and NetworkManagerHUD components.
- A GUI element is created to allow new messages to be typed in. Add a UI/Input Field to the scene.
- The notice board itself is a 3D Text object. A script component called UpdateText is added to this, with the initial code shown in Algorithm 70 on the next page which provides a publicly accessible onTextAdded event handler for when new text needs to be added. The script does rely on the name of the input field remaining as the default value of “InputField”.

It should be possible to run the application as a host at this point, and (ignoring error messages about lack of player prefabs) enter text into the input field and see it appended to that present on the notice board.

2. The next step is to convert the notice board into a prefab that is spawned on the server, and for which imposters are then shared on all clients.

Add a NetworkIdentity component to the notice board so that it can be used as a networked object.

Convert the notice board to a prefab, and remove it from the scene.

In the NetworkManager, add the notice board prefab to the list of Registered Spawnable Prefabs.

Algorithm 70 Transferring text messages from an input field and appending them to a notice board.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5
6  public class UpdateText : MonoBehaviour {
7
8      // Use this for initialization
9      void Start () {
10         InputField inputField = GameObject.Find (""
11             InputField).GetComponent <InputField> ();
12         inputField.onEndEdit.AddListener (onTextAdded);
13     }
14
15     public void onTextAdded (string message)
16     {
17         GetComponent <TextMesh> ().text += "\n" + message;
18     }
  
```

Create a new empty object called MessageBoardGenerator, and add the script in Algorithm 71 on the facing page to a C# component named MakeMessageBoard. This spawns a new message board object when it is run on the server. This ensures that the server then creates message board imposters on each of the clients. At this point each instance of the application on a different device should be able to enter text in the input field and see this appear on the local message board object. The next step is to ensure that this change is sent to the server, which then relays it to each client.

3. The response to the text added event is changed to send this as a notification to the server. The client must not manipulate the local copy of the message board directly.

```

1  public void onTextAdded (string message)
2  {
3      CmdOnTextAdded (message);
4  }
  
```

Algorithm 71 Spawning of a shared object is achieved through an NetworkBehaviour derived script, ensuring that the spawn operation is only run on the server.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.Networking;
5
6  public class MakeMessageBoard : NetworkBehaviour {
7      public GameObject messageBoardPrefab;
8
9      void Start ()
10     {
11         if (isServer)
12         {
13             GameObject board = Instantiate (
14                 messageBoardPrefab);
15             NetworkServer.Spawn (board);
16         }
17     }

```

4. This is received on the server:

```

1  [Command]
2  void CmdOnTextAdded (string message)
3  {
4      messageChange = message;
5 }

```

5. The server modifies a local sync variable, which when changed invokes a function on each client (including the originator of the message) and that updates the content on the local message board object.

```

1  [SyncVar (hook = "onSyncTextChanged")]
2  private string messageChange;
3
4  void onSyncTextChanged (string message)
5  {
6      GetComponent <TextMesh> ().text += "\n:" + 
7          message;

```

The Local Player Authority must be set on the NetworkIdentity component of the message board, so that

this object has the necessary permissions to modify itself across the network.

6. However this solution is confounded by lack of options to allow objects to update themselves without actually being player objects. The client to server remote procedure call can be replaced with a direct message option.

Firstly we define a message class that contains the information that the client needs to send to the server. In this case, it just consists of the message string entered into the input field.

```

1  private short MessageBoardMessageType = MsgType.
   Highest + 1;
2  class MessageBoardMessage : MessageBase
3  {
4      public string messageUpdate;
5 }
```

The message is identified by a unique message ID number stored in the variable *MessageBoardMessageType*, which needs to have a value unique across the whole application. Other types of message that get sent must be assigned different numbers.

7. Instead of invoking the command function, we instead directly send a message to the server:

```

1  MessageBoardMessage networkMessage = new
   MessageBoardMessage ();
2  networkMessage.messageUpdate = message;
3  NetworkClient.allClients[0].Send (
   MessageBoardMessageType, networkMessage);
```

This code does assume the active client connection is the first on the list, which may not be true in more complex configurations.

8. The server then needs to watch out for messages of this particular type and redirect them to another function. This happens in the Start function when the message board is created.

```

1  if (isServer)
2  {
3      NetworkServer.RegisterHandler (
        MessageBoardMessageType, MessageOnTextAdded);
4 }

```

9. The function `MessageOnTextAdded` is now called as an event handler every time a message of this type arrives. It needs to extract the message string on the server, and can then resume the pattern described previously by updating the synchronized variable (which distributes the update to all clients). The server is allowed to send updates to non-player clients.

```

1  void MessageOnTextAdded (NetworkMessage m)
2  {
3      messageChange = m.ReadMessage <
        MessageBoardMessage> ().messageUpdate;
4 }

```

10. The complete script for this process is provided in Algorithms 72 on the next page and 73 on page 291. The view of the conversion from the point of view of one of the clients is shown in Figure 8.3.2.

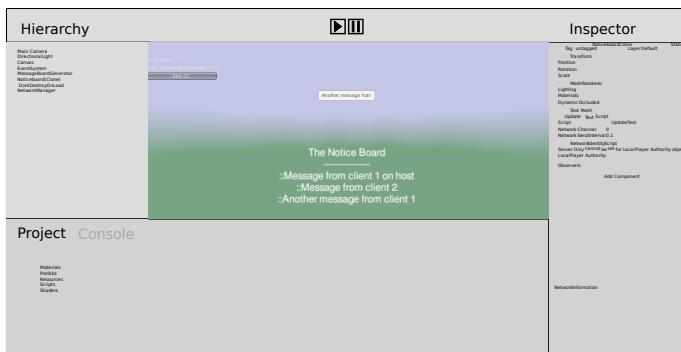


Figure 8.3.2: Output from two clients each updating a message board which then shares updates with each of its instances on each client.

Algorithm 72 A network object can provide an interface allowing custom properties to be updated, and distribute those changes to all other versions of itself (imposters) on all clients (part 1).

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5  using UnityEngine.Networking;
6
7  public class UpdateText : NetworkBehaviour {
8      void Start () {
9          InputField inputField = GameObject.Find (""
10             InputField").GetComponent <InputField> ();
11             inputField.onEndEdit.AddListener (onTextAdded);
12
13         if (isServer)
14         {
15             NetworkServer.RegisterHandler (
16                 MessageBoardMessageType, MessageOnTextAdded
17             );
18         }
19     }
20
21     [SyncVar (hook = "onSyncTextChanged")]
22     private string messageChange;
23
24     void onSyncTextChanged (string message)
25     {
26         GetComponent <TextMesh> ().text += "\n:" + message;
27     }
28
29     [Command]
30     void CmdOnTextAdded (string message)
31     {
32         messageChange = message;
33     }
34
35     void MessageOnTextAdded (NetworkMessage m)
36     {
37         messageChange = m.ReadMessage <MessageBoardMessage
38             > ().messageUpdate;
39     }
40     // continued ...
```

Algorithm 73 A network object can provide an interface allowing custom properties to be updated, and distribute those changes to all other versions of itself (imposters) on all clients (part 2).

```
1  // continuation ...
2  public void onTextAdded (string message)
3  {
4      MessageBoardMessage networkMessage = new
5          MessageBoardMessage ();
6      networkMessage.messageUpdate = message;
7      NetworkClient.allClients[0].Send (
8          MessageBoardMessageType, networkMessage);
9
10     private short MessageBoardMessageType = MsgType.
11         Highest + 1;
12     class MessageBoardMessage : MessageBase
13     {
14         public string messageUpdate;
15     }
16 }
```

9

Dynamic Geometry

9.1 Procedural Content Creation

9.1.1 Description

Procedural control is relevant in many aspects of virtual reality application development. Controlling behaviour of autonomous objects, and customizing responses to user interactions are common reasons to develop custom programs as part of the application development process. A range of other opportunities are achieved through procedural content creation processes.

Consider the level of detail and variety that should actually be present in a virtual world. In a natural landscape, every leaf, blade of grass, rock and grain of sand is unique. Typical applications would gloss over this detail because the limited performance of current systems directs resources to where they would be most beneficial, and because the human effort in manually modelling millions of similar structures would not be justifiable.

Procedural content creation addresses these issues. A function that generates a tree would be able to determine the level of detail required and produce geometry at that level, adapting the complexity of the virtual world to the computational resources available. That function could also accept parameters (for example, based on position) that would allow it to generate elements that

are equivalent but visually distinctive.

Having embraced this principle further opportunities present themselves. Procedural generation of structures allows the objects created to include additional meta-data relating to their structure. A tree generator, for example, now need not only result in a soup of coloured polygons (which is the typical output of a generic 3D modelling package) but instead a hierarchical structure that identifies regions labelled variously a trunk, branch and leaf. This allows actions such as “blowing in the wind” which might operate on branches, but not the trunk (hypothetically), or “rustling” which might just modify the leaves. Extra parameters included in such meta-data might include thickness of a branch, or attached leaf area, which would affect wind response.

In an urban situation, buildings would be created out of rooms, which might in turn have several walls, windows and doors. Several types of window may exist and these are selectively reused to provide variety. The type of room would also affect the choice of furnishing which would be placed procedurally using an understanding of the role of each item. Meta-data resulting from this process could include navigation information, identifying where avatars could move. Details of the items present would support the development of scripts to allow autonomous entities to correctly interact with the room elements.

A key element of procedural generation that we exploit for this pattern is the reuse of content generators. Decomposing the task of content generation uses the same abstraction skills that we used to break down complex processes into individual functions (section 3.5 on page 104) and gives all the benefits of reuse (call the same function in multiple different ways), variety (pass parameters to the function to modify its output), meta-data labelling (the name of the function and its parameters constitute meta-data), and control (issues with the final result can be resolved by modifying individual

functions and regenerating the result).

9.1.2 *Pattern*

Objects are procedurally generated using a top-down approach starting with a function whose task is to create the complete object. This function would identify the components of the object, and call further functions to generate each of these in turn. This process is repeated until the functions are simple enough to return individual geometry elements (either primitives such as cubes or spheres, or non-decomposable elements that are manually modelled in a 3D-modelling package).

Each function would take a number of parameters. These would typically include properties such as relative position, or size and shape. Parameters should be chosen to be descriptive of the object being created (number of floors being more useful for creating meaningful skyscrapers than height in meters).

It is very tempting to use (pseudo)random numbers within a function to provide the variation in the output content. This should be avoided. Ideally these functions are deterministic so that they return the same output for a given set of parameters. This ensures consistency if content needs to be recreated or refined during the course of a virtual reality experience. Seeding a random number generator to provide deterministic random numbers is problematic if other parts of the application might be sharing the same random number generator. A recommended approach is to include one or more identifier parameters. These are just extra numeric parameters that act as unique identifiers for a particular instance of the content returned. Random(ish) values can then be produced through use of different hashes of these identifiers, through functions such as Perlin noise.

A typical component generator function would have the form:

```
function generateComponentA (content relevant
    parameters, identifier parameter)
{
    childIdentifier1 =
        hash1 (identifier parameter)
    childIdentifier2 =
        hash2 (identifier parameter)

    ...
    childContentParameter1 =
        f1 (content parameters)
    childContentParameter2 =
        f2 (content parameters)

    ...
    childComponentX = generateComponentB
        (childContentParameter1, childIdentifier1)
    childComponentY = generateComponentB
        (childContentParameter2, childIdentifier2)
    ...
    any additional procedural generation code
    that utilizes content and identifier
    parameters
    ...
    return (childComponentX, childComponentY, ...)
}
```

This pattern assumes the content for the component might be produced by some primitive content generation functionality (represented by the additional procedural generation code) combined with content produced by a number of subsidiary content generation functions. We use different hash functions based on the identifier provided to deterministically produce identifiers for each of the child components. This way, even if we use the same function to produce each of the children, the content produced for each can still be slightly different to the extent supported by this function. For example,

a function to produce a tree might create the geometry for the trunk, but then call a branch function to produce several distinctive branches which could be placed along the trunk.

9.1.3 Example

Example 38. Applying the pattern to Unity software

1. This example describes the process of procedurally generating a building. Buildings typically consist of walls, which in turn are made up of sections with doors and windows. Ultimately each of these is then built in turn from individual bricks.

To provide a little extra challenge with this example, the bricks have the physics simulation enabled on them ensuring that the resulting structures can be modified (mostly destroyed) by physical interactions with other objects.

Start with a brick. A cube primitive object is a useful initial shape. Leave the default shape, since the building generator assumes that each brick has unit dimensions, and an object origin in the center of the brick. Add a rigidbody component to provide the physics effects, leaving gravity enabled. Add any appropriate material, before converting the object into a prefab.

A ground plane is also advisable so that the building has a surface to rest on.

2. Buildings are created by a building manager object. This is an empty object, with a C# HouseBuilder script attached.
3. A building consists of a number of floors stacked vertically, capped with a roof. This is represented with following pseudo-code:

```
1  function building (size, shape, number of floors)
2      for each floor
```

```

3      generate floor structure
4      generate roof structure

```

4. In this case, one floor of the building consists of four walls, and a flat roof slab to hold the roof or any subsequent floors. We can add some variety by including different types of wall, either solid, solid with a door, or solid with one or more windows. The pseudo-code for this looks like:

```

1  function floor (size, shape)
2    generate wall with door
3    generate wall with window
4    generate solid wall
5    generate solid wall
6    generate roof slab

```

5. A solid wall involves just piling up bricks into a rectangular region. We can be a bit more specific about the size and shape parameters now. To produce walls in any direction, we might specify a vector H in the horizontal direction of the wall, and another vector V aligned with the vertical direction of the wall. The size of the wall might be h bricks along by v bricks up. The wall can start at position p , and extend to position $p + b(hH + vV)$, where b is the size of one brick.

```

1  function wall (p, H, V, h, v, b)
2    for (i = 0; i < v; i++)
3      for (j = 0; j < h; j++)
4        place brick at p+b*(i*V+j*H)
5        set size of brick to b

```

6. Walls with doors and windows are created from sub-walls. For example a wall with a window in the center might involve:

```

1  function wallWithWindow (p, H, V, h, v, b)
2    // first third
3    wall (p, H, V, h/3, v, b)
4      // wall below window.
5    wall (p+b(h/3)H, H, V, h/3, v/3, b)
6    // wall above window
7    // note: a solid beam may be required as a lintel
          to hold the upper wall.

```

```

8   wall (p+b((h/3)H+(2v/3)V), H, V, h/3, v/3, b)
9   // final third of wall
10  wall (p+b(2h/3)H, H, V, h/3, v, b)

```

7. A pitched roof is created from slabs, each positioned one about the other, and shrinking the dimensions of successive slabs. Assume that the base of the roof is h by w bricks in size.

```

1  function pitchedRoof (p, H, V, h, w, b)
2  while h > 0 and w > 0
3    createSlab (p, H, V, h, w, v, b)
4    h = h - 2b
5    w = w - 2b
6    v = v + b

```

8. A slab and a brick are created from the brick prefab. The slab is scaled to the size required, and the brick is scaled to the dimensions provided by the parameter for the brick size.
9. The complete script for generating a building is provided in Algorithms 74, 75, 76, 77, 78, 79, 80, 81 and 82. An example of a building produced using this process is shown in Figure 9.1.1. Tall structures need to have gravity disabled on the individual bricks to remain stable under the physics simulation. Turning on gravity produces effects such as those shown in Figure 9.1.2. This process can be extended to create an entire city generated using the building model with different parameters. The entire structure is vulnerable to users manipulating collision generating tools.

9.2 Dynamic Patterns

9.2.1 Description

Textures are frequently used to add detail to virtual worlds. Their most common incarnation is as a 2D image used to add colour patterns to surfaces but the concept of a texture is far more flexible. Even the 2D image version is regarded as a function that converts some coordinates

Algorithm 74 Building construction part 1

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class HouseBuilder : MonoBehaviour {
6      public GameObject brickPrefab;
7      public float brickSize = 0.5f;
8      public float length = 5.0f;
9      public float width = 3.0f;
10     public float height = 2.0f;
11     public int floors = 5;
12
13     // continued ...
```

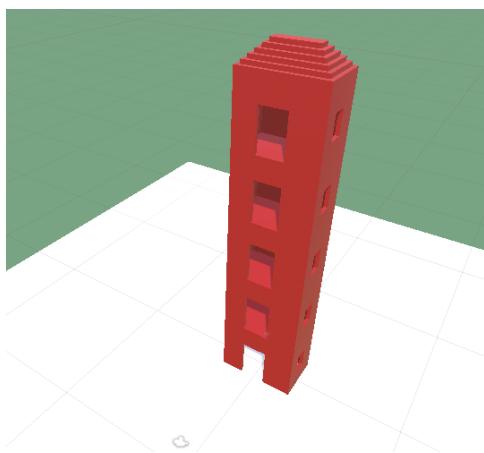


Figure 9.1.1: A view of the procedurally generated building.

Algorithm 75 Building construction part 2

Algorithm 76 Building construction part 3

```
1 // Create a single object wall, required for
   structural strength.
2 private void buildLintel (Vector3 start, Vector3
   hdirection, Vector3 hDirection, float length,
   float height, float brickSize, GameObject
   brickPrefab)
3 {
4     Vector3 brickPos = start + (length / 2) * hdirection
       + (height / 2) * hDirection + (brickSize / 2)
       * (new Vector3 (1,1,1) - (hdirection +
       hDirection));
5     GameObject brick = Instantiate (brickPrefab,
       brickPos, Quaternion.identity);
6     brick.transform.localScale = (length * hdirection +
       height * hDirection + brickSize * (new Vector3
       (1,1,1) - (hdirection + hDirection)));
7     brick.transform.SetParent (this.transform, false);
8 }
```

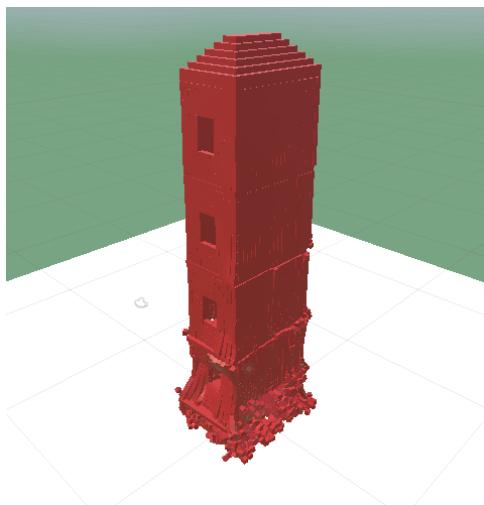


Figure 9.1.2: A hint of the opportunities offered by generating complex structures procedurally.

Algorithm 77 Building construction part 4

```
1  private void buildWallWithDoor (Vector3 start, Vector3
        hdirection, Vector3 hDirection, float length,
        float height, float doorWidth, float doorHeight,
        float brickSize, GameObject brickPrefab)
2  {
3      float doorSize = (Mathf.Round (doorWidth / brickSize
            )) * brickSize; // make sure door is a whole
            number of bricks in size.
4      float doorStart = (Mathf.Round ((length - doorSize)
            / (2 * brickSize))) * brickSize; // and that
            the door starts at a whole brick.
5      float doorTop = (Mathf.Round (doorHeight / brickSize
            )) * brickSize; // make sure door is a whole
            number of bricks in height.
6
7      // left half
8      buildWall (start, hdirection, hDirection, doorStart,
            doorTop, brickSize, brickPrefab);
9      // right half
10     buildWall (start + new Vector3 (doorStart + doorSize
            , 0, 0), hdirection, hDirection, length - (
            doorStart + doorSize), doorTop, brickSize,
            brickPrefab);
11
12     // lintel
13     buildLintel (start + new Vector3 (0, doorTop, 0),
            hdirection, hDirection, length, brickSize,
            brickSize, brickPrefab);
14
15     // Layer of bricks above lintel
16     buildWall (start + new Vector3 (0, doorTop +
            brickSize, 0), hdirection, hDirection, length,
            height - (doorTop + brickSize), brickSize,
            brickPrefab);
17 }
```

Algorithm 78 Building construction part 5

```
1  private void buildWallWithWindow (Vector3 start,
2          Vector3 hdirection, Vector3 hDirection, float
3          length, float height, float windowHeight, float
4          windowBottom, float windowTop, float brickSize,
5          GameObject brickPrefab)
6  {
7      float windowHeight = (Mathf.Round (windowWidth /
8          brickSize)) * brickSize;
9      float windowStart = (Mathf.Round ((length -
10         windowHeight) / (2 * brickSize))) * brickSize;
11     windowBottom = (Mathf.Round (windowBottom /
12         brickSize)) * brickSize;
13     windowTop = (Mathf.Round (windowTop / brickSize)) *
14         brickSize;
15
16     // bottom layer
17     buildWall (start, hdirection, hDirection, length,
18                 windowHeight, brickSize, brickPrefab);
19
20     // left half
21     buildWall (start + windowBottom * hDirection,
22                 hdirection, hDirection, windowStart, windowTop
23                 - windowHeight, brickSize, brickPrefab);
24
25     // right half
26     buildWall (start + (windowStart + windowHeight) *
27                 hdirection + windowHeight * hDirection,
28                 hdirection, hDirection, length - (windowStart +
29                     windowHeight), windowTop - windowHeight,
30                     brickSize, brickPrefab);
31
32     // lintel
33     buildLintel (start + windowTop * hDirection,
34                  hdirection, hDirection, length, brickSize,
35                  brickSize, brickPrefab);
36
37     // Layer of bricks above lintel
38     buildWall (start + (windowTop + brickSize) *
39                 hDirection, hdirection, hDirection, length,
40                 height - (windowTop + brickSize), brickSize,
41                 brickPrefab);
42 }
```

Algorithm 79 Building construction part 6

```
1 // a pitched roof consists of a number of slabs of
   decreasing size, stacked one about the other.
2 private void buildRoof (Vector3 start, Vector3
   lDirection, Vector3 wDirection, Vector3
   hDirection, float length, float width, float
   brickSize, GameObject brickPrefab)
3 {
4     float l = length;
5     float w = width;
6     float h = 0;
7     while ((l > 0) && (w > 0))
8     {
9         buildRoofSlab (start + h * (hDirection +
           lDirection + wDirection), lDirection,
           wDirection, hDirection, l, w, brickSize,
           brickPrefab);
10    //     GameObject brick = Instantiate (brickPrefab
11    // , brickPos, Quaternion.identity);
12    //     brick.transform.localScale = (l *
13    // lDirection + w * wDirection + brickSize *
14    // hDirection);
15    //     brick.transform.SetParent (this.transform,
16    // false);
17 }
```

Algorithm 8o Building construction part 7

```

1 // create a slab of length by width bricks with height
   of 1 brick.
2 private void buildRoofSlab (Vector3 start, Vector3
   lDirection, Vector3 wDirection, Vector3
   hDirection, float length, float width, float
   brickSize, GameObject brickPrefab)
3 {
4     float l = length + brickSize;
5     float w = width + brickSize;
6     float h = brickSize / 2;
7     Vector3 brickPos = start + (l / 2) * lDirection + (w
       / 2) * wDirection + h * hDirection;
8     GameObject brick = Instantiate (brickPrefab,
       brickPos, Quaternion.identity);
9     brick.transform.localScale = (l * lDirection + w *
       wDirection + brickSize * hDirection);
10    brick.transform.SetParent (this.transform, false);
11 }

```

Algorithm 81 Building construction part 8

```

1 // A floor consists of 4 walls (one with door, and one
   with window, and a flat roof slab.
2 private void buildOneFloor (Vector3 start, Vector3
   lDirection, Vector3 wDirection, Vector3
   hDirection, float length, float width, float
   height, float brickSize, GameObject brickPrefab)
3 {
4     // 4 walls
5     buildWallWithDoor (start, lDirection, hDirection,
       length, height, length / 3.0f, height * 2.0f /
       3.0f, brickSize, brickPrefab);
6     buildWallWithWindow (start + length * lDirection,
       wDirection, hDirection, width, height, width /
       3.0f, height * 1.0f / 3.0f, height * 2.0f /
       3.0f, brickSize, brickPrefab);
7     buildWall (start + length * lDirection + width *
       wDirection, -lDirection, hDirection, length,
       height, brickSize, brickPrefab);
8     buildWall (start + width * wDirection, -wDirection,
       hDirection, width, height, brickSize,
       brickPrefab);
9
10    buildRoofSlab (start + height * hDirection,
       lDirection, wDirection, hDirection, length,
       width, brickSize, brickPrefab);
11 }

```

Algorithm 82 Building construction part 9

```

1   // A house consists of a number of floors, and a
2   // pitched roof.
3   private void buildHouse (Vector3 start, float length
4       , float width, float wallheight, float
5       brickSize, GameObject brickPrefab)
6   {
7       // assert: length, width, height must be a
8       // multiple of bricksize.
9       length = (Mathf.Round (length / brickSize)) *
10      brickSize;
11      width = (Mathf.Round (width / brickSize)) *
12      brickSize;
13      height = (Mathf.Round (wallheight / brickSize)) *
14      brickSize;
15
16      Vector3 lDirection = new Vector3 (1, 0, 0);
17      Vector3 wDirection = new Vector3 (0, 0, 1);
18      Vector3 hDirection = new Vector3 (0, 1, 0);
19
20      for (int i = 0; i < floors; i++)
21      {
22          buildOneFloor (start + (i * (height + brickSize)
23              ) * hDirection, lDirection, wDirection,
24              hDirection, length, width, height,
25              brickSize, brickPrefab);
26      }
27
28      // roof
29      buildRoof (start + ((floors - 1) * (height +
30          brickSize) + brickSize) * hDirection + height
31          * hDirection, lDirection, wDirection,
32          hDirection, length, width, brickSize,
33          brickPrefab);
34  }
35
36  void Start () {
37      buildHouse (new Vector3 (-length / 2, 0, -width /
38          2), length, width, height, brickSize,
39          brickPrefab);
40  }
41 }
```

(u, v) to the colour value of the texture at those coordinates.

Mathematical functions serve as a way of transforming some input parameters into particular output that has significance to us. We can write this as:

$$y = f(x_1, x_2, \dots, x_n)$$

This describes how we get some valuable information y by modifying inputs x_1, x_2 up to x_n through the process described through function f . However in the context of virtual reality we may run into some limitations:

- Computers have limited accuracy and so each input x_i no longer represents a continuous space but rather samples at discrete (possibly very small) intervals. However we have now been required to accept that samples are a trade-off we're prepared to make.
- The function f may be very difficult to compute and it may be far easier just to measure it from similar processes that may exist in the physical world at our defined sample points. Examples include photographs, but also bidirectional reflectance distribution functions (BRDFs) that capture the interaction of light with a surface under a range of different configurations.
- The function f may be slow to compute. We might have a way of procedurally generating a particular effect but it is slow. We can then compute values at interesting sample points, and store the results in a big lookup table. Retrieving data from the lookup table involves accessing the elements in the (multi-dimensional) array using code such as:

$$y = f_{lookup}[x_1, x_2, \dots, x_n]$$

You might notice how using a function and accessing a texture lookup table are only cosmetically different, and achieve the same underlying goals.

Textures provide us with a lookup table, indexed by as many input parameters as we need, and capable of returning any value that might be meaningfully computed. The difference between procedural generation and texture lookup is only due to pragmatic issues around performance and storage requirements. For ease of storage and lookup, the input parameters in a texture are sampled at regular intervals so that a multi-dimensional array can be used (although other options are considered where the information in the texture is sparse (e.g. mostly zero) or could be easily compressed).

The patterns in a procedurally generated texture are often composed of a number of known primitive functions with easily manipulated properties. With respect to the common application of texture for representing coloured images, the typical Cartesian view of the function $y = f(x)$ uses x as the spatial dimension across the width/height of the texture, and y is related to the colourfulness (or brightness) of the colour stored at each texel in the texture. Properties that are relevant in this context are:

- frequency: how often elements of the pattern repeat, or inversely, what the scale of features in the texture image are.
- amplitude: what the range of brightness levels are.

9.2.2 *Pattern*

The patterns covered in this section include a standard template for defining and populating a texture lookup table, and some common primitive functions used to create particular categories of texture pattern.

The texture is represented by an array, a common collection structure in most languages which maps directly into a memory efficient structure easily used on virtual reality rendering hardware. For the sake of generality, a n -dimensional texture pattern is shown, although most image based uses of textures are limited to 2 dimensions.

```

// declare texture.
Texture texture[Size1, Size2, ..., SizeN]
// populate texture from a function
for (x1 = 0; x1 < Size1; x1++)
{
    for (x2 = 0; x2 < Size2; x2++)
    {
        ...
        for (xn = 0; xn < SizeN; xn++)
        {
            texture[x1, x2, ..., xn] =
                f (x1, x2, ..., xn)
        }
    }
}

```

The number of samples (Size) for each of the dimensions needs to be known in advance. The values in the texture are taken from function f in this case, allowing the texture to use the procedural generation process defined by this function. Another common approach is to store the values in a file, and read them out into the texture one by one. The values in the file may result from data captured in the physical world (e.g. photograph) or from some previous slow procedural generation process that was then written to the file for later retrieval. The texture array may need to be converted into an engine specific format for use in the graphics system but most engines provide functions that convert to and from the array representation.

The nesting of the for loops as shown in this pattern is a common strategy for accessing all elements in regular multi-dimensional structures such as our texture arrays. Each inner loop is repeated for every iteration of the outer loop. This is similar to the scanning process used by early cathode ray based televisions; the beam would

iterate over each row, and while in a row would iterate over each column.

Patterns with regular repeated features are periodic. Periodic patterns are well suited to procedural generation using periodic functions. The functions sin and cos are two of the better behaved periodic functions and known to be versatile functions that can be combined (using a variety of amplitudes and frequencies) to construct other patterns. The pattern for a single periodic pattern is:

```
function f (x1, x2, ..., xn, A, f, ϕ)
{
    return A * sin (f * xi + ϕ)
}
```

This version uses only one of the input parameters, x_i . It also allows control of the pattern's properties through parameters A (amplitude), f (frequency) and ϕ (phase). The phase is used to offset one pattern relative to another.

The pattern is manifested visually as repeated structures (lines in a 2D texture) running perpendicular to dimension i . Including additional x parameters at various points in the template can affect the direction of the pattern (add to the x_i term), modulate different patterns (include more sin terms multiplying the one shown) or composited (add more sin terms). Amplitude can also be a colour value, if particular colour values are required, although more control is achieved by using the sin term as a blending factor in an interpolation pattern (section 10.3 on page 347).

Irregular or organic patterns should not be produced using random numbers. Apart from difficulties in consistently recreating random fields, random patterns have very high frequency components which are not compatible with the sampling processes that occur in texturing. A frequency limited noise function, such as Perlin noise [Perlin, 1985, 2002], provides the required varied and natural appearance.

```

function f (x1, x2, ..., xn, A, f, ϕ)
{
    return A * noise
    (f * xi + ϕ,
     f * xj + ϕ,
     ...,
     f * xk + ϕ)
}

```

Multidimensional versions of noise functions exist and are used to achieve patterns that vary over a number of the input dimensions. As in previous patterns, the amplitude, frequency and phase parameters are also meaningful in this context as they relate to properties of the texture features even though the features are not exactly repeated as they are in periodic patterns.

9.2.3 Example

Example 39. Applying the pattern to Unity software

1. The example demonstrates how textures are procedurally generated in Unity software.

Create an object. The texture is applied to the surface of this object, so we can evaluate the effect of the pattern generation process.

Create a C# script named TextureGenerate, and add it as a component to the object you have created.

2. The process of generating a texture involves creating an image, and writing a colour value to each pixel in that image. There are only a few other housekeeping details required, such as applying the changes to the texture once all the modifications are complete (this avoids the time consuming process of having to update internal structures in the virtual reality engine and graphics card after every pixel write), and copying the texture to the appropriate property of the material applied to the object.

The code for achieving this is shown in Algorithm 83.

3. The particular pattern to be used is produced in a separate function. This function takes the coordinates of a point (u, v) within the pattern space, and returns the corresponding colour.

A periodic pattern is produced using a sin function, using a combination of the u and v values as the parameter purely to get the stripes at an angle.

```

1  public float frequency = 5.0f;
2  public float phase = 0.0f;
3  public float amplitude = 1.0f;
4
5  Color stripes (float u, float v)
6  {
7      float brightness;
8      brightness = (amplitude * Mathf.Sin (frequency *
9          (0.3f * u + 0.5f * v) + phase) + 1.0f) / 2.0f
10     ;
11
12     return new Color (brightness, brightness,
13         brightness);
14 }
```

Experiment with the frequency, phase and amplitude parameters to appreciate how they affect the nature of the pattern produced.

4. Organic patterns make use of Perlin noise.

```

1  public float frequencyX = 5.0f;
2  public float frequencyY = 5.0f;
3
4  Color noise (float u, float v)
5  {
6      float brightness = amplitude * Mathf.PerlinNoise
7          (frequencyX * u + phase, frequencyY * v +
8              phase);
9      return new Color (brightness, brightness,
10         brightness);
11 }
```

On its own it is not terribly exciting, but it is worth modifying the frequency parameters shown in Figure 9.2.1 to get a feel of the ways the pattern can adjust to match structures associated with surface patterns of various scales.

Algorithm 83 Texture generation involves assigning a colour from an appropriate pattern function to each pixel in an image.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TextureGenerate : MonoBehaviour {
6
7      void On	TextureGenerate () {
8          {
9              int texWidth = 256;
10             int texHeight = 256;
11
12             Texture2D synthTexture = new Texture2D (texWidth,
13                                         texHeight);
14
15             for (int i = 0; i < texWidth; i++)
16             {
17                 for (int j = 0; j < texHeight; j++)
18                 {
19                     float u = 1.0f * i / texWidth;
20                     float v = 1.0f * j / texHeight;
21                     Color col = stripes (u, v);
22                     // Color col = noise (u, v);
23                     // Color col = wood (u, v);
24                     synthTexture.SetPixel (i, j, col);
25                 }
26
27             synthTexture.Apply ();
28
29             GetComponent <Renderer> ().materials[0].SetTexture
30                         ("_MainTex", synthTexture);
31
32             void Start () {
33                 On_TextureGenerate ();
34             }
35 }
```

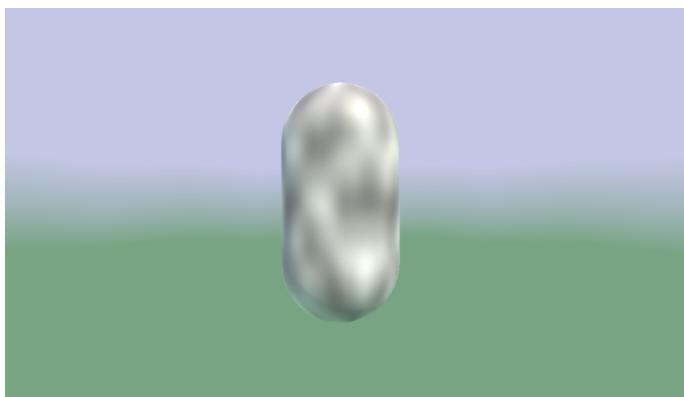


Figure 9.2.1: Perlin noise pattern produced during texture synthesis.

5. A combination of stripes and noise sourced irregularity can start to model the varied patterns found in the physical world. For example a wood grain style texture as shown in Figure 9.2.1 is produced by mixing colours according to a distorted stripe pattern.

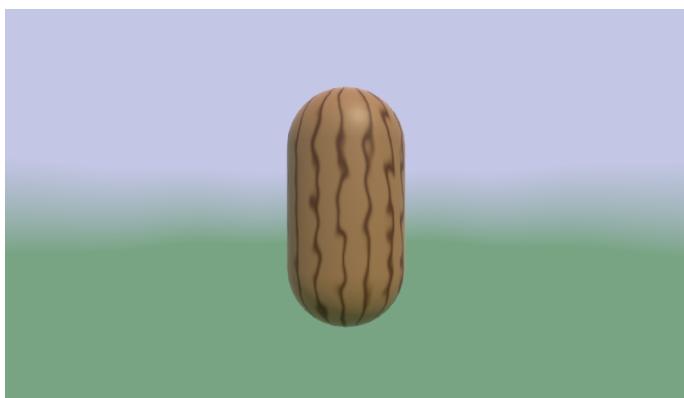


Figure 9.2.2: Wood grain pattern generated using a combination of periodic patterns and Perlin noise.

```

1 public float stripeFrequency = 100.0f;
2 public float seed = 4352.32423f;
3 public float grainFineness = 10.0f;
4 public float wriggleFrequencyU = 30.0f;
5 public float wriggleFrequencyV = 10.0f;
6 public float wriggleAmplitude = 0.07f;
7
8 public Color woodColor = new Color (150.0f / 255.0f
   , 111.0f / 255.0f, 70.0f / 255.0f);
9 public Color grainColor = new Color (80.0f / 255.0f
   , 41.0f / 255.0f, 0.0f / 255.0f);
10

```

```

11  Color wood (float u, float v)
12  {
13      float wriggle = wriggleAmplitude * Mathf.
14          PerlinNoise (wriggleFrequencyU * u + seed,
15              wriggleFrequencyV * v + seed);
16      //return Color (wriggle, wriggle, wriggle);
17
18      float stripe = 0.5f + 0.5f * Mathf.Sin (
19          stripeFrequency * (u + wriggle) + seed);
20      stripe = Mathf.Pow (stripe, grainFineness);
21      //return Color (stripe, stripe, stripe);
22
23      return (1.0f - stripe) * woodColor + stripe *
24          grainColor;
25  }

```

The parameters for this pattern generator include a seed variable which provides an offset in pattern space. This allows unique variations of the pattern while still retaining all of the appearance characteristics defined by the other parameters. Thus multiple textures are produced that are equivalent but distinct.

6. The texture patterns synthesized can also be used for other effects. For example, including the lines below at the end of the `On_TextureGenerate` function assigns the texture to the property used for bump/normal mapping.

```

1 GetComponent <Renderer> ().materials[0].SetTexture
2     ("_BumpMap", synthTexture);
3 GetComponent <Renderer> ().materials[0].
4     EnableKeyword ("_NORMALMAP");

```

9.3 Shaping up

9.3.1 Description

In addition to generating structures out of primitive shapes and other pre-prepared content (section 9.1 on page 293) it is also possible to generate and manipulate the shape and structure of the actual geometric mesh used for individual objects. This provides all the benefits of procedural generation with the added advantage of

being able to manipulate the shape of objects dynamically. For example, a procedural generation process can generate a mesh for the terrain underfoot. This geometry can be changed to reflect any changes engineered by the users (digging, or flattening in preparation for building, for example). It is even possible to adapt the terrain as the user moves, adding more in front and removing from the back, so that a small and efficient mesh can appear to contain a vast and wondrous landscape.

The mesh is a boundary representation so only describes the surface of objects. The interior of such objects is empty, or even invisible if we follow the common convention of only drawing the outer surface. This is a convenient representation for current graphics architectures although we may see a change to representations better suited to the expectations of virtual reality environments in future.

A typical mesh structure is made up of a number of components:

Vertex: The vertices of the mesh represent the set of points that lie on the surface of an object. Most meshes are made of polygonal (usually triangular) faces and so the surface is approximated using these points which lie on the corners of the faces. The information associated with a vertex must include the position of the vertex, stored as the 3 Cartesian coordinate values: x, y and z. Additional properties of the vertex include the normal direction (the vector pointing perpendicularly outwards from the surface at this point), the tangent direction (a vector parallel to the surface), or texture coordinates (e.g. which point in an image texture is mapped to this vertex).

Faces: The relationships (connections) between vertices are required to transform the point cloud of vertices into a coherent surface. A face consists of a sequence of vertices (3 for triangles) identified by some unique key such as their index in the list of vertices.

The order in which the vertices are listed (winding) is crucial, and can be either clockwise or anti-clockwise as viewed from outside the surface. The particular ordering convention employed depends on the platform used, and determines which surface of a face is considered to be the outside (normal direction). Procedural mesh generators must ensure all faces are generated according to a consistent winding strategy. Winding order is maintained by making sure that the two faces that share an edge must list them in the opposite order (i.e. one defines edge as A to B, the other as B to A).

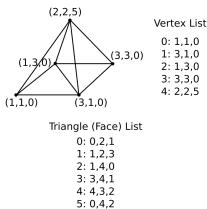


Figure 9.3.1: Geometry in a virtual world is represented as a list of vertices, and a list of faces described through reference to the list of vertices.

Faces may also have other properties. For example texture coordinates may be specified for each vertex of a face. This would allow multiple texture images to be used on a single mesh. Texture coordinates in this case would also be defined by referencing their index in a list of texture coordinate values.

Procedural generation of meshes involves building up the list of vertices, describing how they are connected by defining a list of faces, and then creating or replacing these properties in the mesh component of an object. This would then be available to be rendered for the next frame.

Mesh structure can also be modified dynamically in a shader (section 10.1 on page 329) during the rendering process. This form of procedural modification would start with a pre-prepared mesh since vertex and geometry shaders only modify the properties of existing vertices, or refine them based on a initial configuration provided to the graphics pipeline.

9.3.2 Pattern

Mesh creation proceeds by first defining the list of vertices. Most engines prefer to receive these as an array of vertices. We may need to define which properties we want in a vertex. The structure below contains some of

the typical vertex properties but also some additional options that may be useful for particular situations.

```
class Vertex
{
    // Common properties
    Vector position
    Vector normal
    Vector textureCoordinates
    // Less common properties
    Vector tangent
    Colour colour
    float height
}
```

The array of vertices is easiest to create if it is possible to predict the number of vertices involved in advance.

```
Vertex vertices [numberOfVertices]
```

This is then populated with the properties of each vertex using a loop.

```
for (i = 0; i < numberOfVertices; i++)
{
    vertices[i].position = Vector (...)

    vertices[i].normal = Vector (...)

    ...
}
```

A common vertex configuration is a grid. We can create a 2D array of vertices for this and use a nested for loop, but since most virtual reality engines prefer the 1D array there are some strategies for achieving the same result. Assume we have a grid with NumberOfRows rows and NumberOfColumns columns.

```
numberOfVertices =
    NumberOfRows * NumberOfColumns
for (i = 0; i < numberOfVertices; i++)
{
```

	Column	0	1	2	3
Row	Index 0	•	•	•	•
1	4	5	6	7	
2	8	9	10	11	
3	12	13	14	15	
4	16	17	18	19	

Number of rows = 5

Figure 9.3.2: A grid of values is represented using a single index (and thus no need to 2D arrays or nested loops) by decomposing the index value into row and column numbers.

```

        row = i / numberColumns
        // using integer division
        column = i % numberColumns
        vertices[i].position = Vector (row, column)
        vertices[i].normal = Vector (...)
        ...
    }
}

```

The integer division/modulus operation provides a way of converting i into two values row and $column$. As i increases, row starts off at 0 and the $column$ increases from 0 until they reach $NumberOfColumns - 1$. Then row increases to 1 and $column$ returns to 0 and starts its cycle again. The process repeats.

An array of faces must also be created. This pattern shows the specific example of triangles - 3 vertex faces. The triangle structure might look like:

```

class Triangle
{
    int v[3]
}

```

The triangle keeps track of the 3 vertices connected to form that triangle. Note that this is not:

```

class Triangle
{
    Vertex v[3]
}

```

We expect that many adjacent triangles may share a vertex. So the value stored in the triangle is the integer representing the index of the vertex in the previous vertex list, not a complete copy of the vertex itself. This also saves time when rendering the mesh, as operations won't have to be repeated each time a vertex is reused in an object.

There are many ways to connect vertices, but the pattern below assumes a common triangle strip approach where each new vertex forms a triangle with the previous 2.

```
numberOfTriangles = numberOfVertices - 2
Triangles t [numberOfTriangles]
for (i = 0; i < numberOfTriangles; i++)
{
    if (i % 2 == 0)
    {
        t[i].v[0] = i
        // index of first vertex of triangle
        t[i].v[1] = i + 1
        // index of second vertex of triangle
        t[i].v[2] = i + 2
        // index of third vertex of triangle
    }
    else
    {
        t[i].v[0] = i + 2
        // index of first vertex of triangle
        t[i].v[1] = i + 1
        // index of second vertex of triangle
        t[i].v[2] = i
        // index of third vertex of triangle
    }
}
```

We need to keep the winding order consist here. The check $i \% 2 == 0$ returns true if i is even, and false if it is odd. Thus every second triangle uses the vertices in the opposite order.

The list of vertices v and the list of faces t are then assigned to properties of the mesh component.

```
mesh.vertices = v
mesh.faces = t
mesh.reload ()
```

The engine may also need to trigger an event to rebuild internal data structures in the virtual reality engine and graphics card.

9.3.3 Example

Example 40. Applying the pattern to Unity software

1. In this example, the mesh supplied with one of the primitive objects is replaced with a procedurally generated mesh. The generated mesh is intended to represent a region of terrain.

Starting with a new Unity software project, add a single primitive object to the scene. Create a C# script component called TerrainGenerator and attach it to this object. Since this object turns into terrain, it is advisable to increase its scale in the horizontal directions.

2. The mesh is defined in terms of the vertices, and the faces (triangles) that connect them. The terrain is represented with a square grid of vertices. Assuming the grid consists of $n \times n$ smaller square faces, then it requires a list of $(n + 1)^2$ vertices to represent it (since we need an extra row and column of vertices to completely enclose the n gaps between vertices). If we refer to the vertex in row i and column j , then we need to work out which element of the list of $(n + 1)^2$ vertices holds the property of that vertex. The expression: $\text{index}(i, j) = i + (n + 1)j$ converts an (i, j) coordinate into the index of the vertex within the list. Similar expressions are used to identify index values for triangles in a list of triangles.

These conversion expressions are convenient to build into functions of their own.

```

1 int indexOfVertex (int x, int y, int sizex, int
                     sizey)
2 {
3     return x + (y * (sizex + 1));
4 }
5

```

```

6 int indexOfTriangle (int x, int y, int sizex, int
    sizey)
7 {
8     return (3 * 2) * (y * sizex + x);
9 }
```

3. The process for generating the terrain mesh is outlined in the following pseudo-code:

```

1 For each (i,j) vertex
2     set the position of the vertex to: (i, heightAt (
        i,j) j)
3     set the texture coordinates of the vertex
4 For each square face linking (i,j) to (i+1,j), (i
        +1,j+1) and (i,j+1)
5     add triangle (i,j) (i+1, j) (i+1,j+1)
6     add triangle (i,j) (i+1, j+1) (i,j+1)
7 Update the vertices in the current object's mesh
8 Update the faces in the current object's mesh
9 Update the normal vectors in the current object's
    mesh
```

The code to achieve this process is shown in Algorithms 84 and 85.

4. The process of finding the height at any particular coordinate is the essence of a terrain synthesis approach. One reasonably versatile approach uses fractal synthesis. This involves overlaying multiple noise functions, each with ever increasing frequency but ever decreasing amplitude. This mimics natural terrain where the size of a feature (inverse of frequency) is proportional to its effect on the height of the terrain. For example: mountains are wide and high, while grains of sand are small and thus high frequency in the horizontal direction and correspondingly add little to the overall height.

A function to implement fractal terrain synthesis is shown in Algorithm 86.

5. Terrain synthesis requires some parameters to control the frequency and amplitude effects. Typical values are:

Algorithm 84 The code to create a mesh representing a grid of vertices and faces. The size of the grid is 250×250 which is close to the maximum size for an individual mesh (part 1).

```

1  void On_UpdateMesh ()
2  {
3      int i;
4      int j;
5
6      int meshx = 250;
7      int meshy = 250;
8
9      Vector3 [] vertices = new Vector3 [(meshx + 1) * (
10         meshy + 1)];
11     for (i = 0; i <= meshx; i++)
12     {
13         for (j = 0; j <= meshy; j++)
14         {
15             float xp = 1.0f * (i - (meshx / 2.0f)) / meshx;
16             float yp = 1.0f * (j - (meshy / 2.0f)) / meshy;
17             vertices[indexOfVertex (i, j, meshx, meshy)] =
18                 new Vector3 (xp, getHeight (xp, yp), yp);
19         }
20     }
21     Vector2 [] uv = new Vector2[(meshx + 1) * (meshy +
22         1)];
23     for (i = 0; i <= meshx; i++)
24     {
25         for (j = 0; j <= meshy; j++)
26         {
27             float xp = 1.0f * (i - (meshx / 2.0f)) / meshx;
28             float yp = 1.0f * (j - (meshy / 2.0f)) / meshy;
29             float u = Mathf.PerlinNoise (patternFrequency *
30                 (xp + seed), patternFrequency * (yp + seed)
31 );
32             float v = 0.3f * (vertices[indexOfVertex (i, j,
33                 meshx, meshy)].y - 0.2f);
34             uv[indexOfVertex (i, j, meshx, meshy)] = new
35                 Vector2 (u, v);
36         }
37     }
38     // continued ...

```

Algorithm 85 The code to create a mesh representing a grid of vertices and faces. The size of the grid is 250 × 250 which is close to the maximum size for an individual mesh (part 2).

```

1   // continuation ...
2   int [] triangles = new int [3 * 2 * meshx * meshy];
3   for (i = 0; i < meshx; i++)
4   {
5       for (j = 0; j < meshy; j++)
6       {
7           triangles[indexOfTriangle (i, j, meshx, meshy) +
8               0] = indexOfVertex (i + 0, j + 0, meshx,
9                   meshy);
8           triangles[indexOfTriangle (i, j, meshx, meshy) +
9               1] = indexOfVertex (i + 0, j + 1, meshx,
10                  meshy);
9           triangles[indexOfTriangle (i, j, meshx, meshy) +
11               2] = indexOfVertex (i + 1, j + 1, meshx,
12                  meshy);
10
11           triangles[indexOfTriangle (i, j, meshx, meshy) +
12               3] = indexOfVertex (i + 0, j + 0, meshx,
13                  meshy);
12           triangles[indexOfTriangle (i, j, meshx, meshy) +
13               4] = indexOfVertex (i + 1, j + 0, meshx,
14                  meshy);
13           triangles[indexOfTriangle (i, j, meshx, meshy) +
15               5] = indexOfVertex (i + 1, j + 1, meshx,
16                  meshy);
14       }
15   }
16
17   Mesh mesh = GetComponent <MeshFilter> ().mesh;
18   mesh.Clear (false);
19   mesh.vertices = vertices;
20   mesh.triangles = triangles;
21   mesh.uv = uv;
22   mesh.RecalculateNormals ();
23 }
```

Algorithm 86 Terrain synthesis using layers of noise.

This formulation is more convenient than other forms of fractal synthesis as the height is determined directly for any horizontal coordinates (x, y) .

```

1  float getHeight (float x, float y)
2  {
3      float h = 0.0f;
4
5      float a = initialAmplitude;
6      float f = initialFrequency;
7
8      for (int l = 0; l < levels; l++)
9      {
10         h = h + a * Mathf.PerlinNoise (f * (x + seed), f *
11             (y + seed));
12         a = a * amplitudeFactor;
13         f = f * frequencyFactor;
14     }
15
16     return h;
17 }
```

```

1  public int levels = 8;
2  public float initialFrequency = 2.0f;
3  public float initialAmplitude = 2.0f;
4  public float frequencyFactor = 2.0f;
5  public float amplitudeFactor = 0.5f;
6  public float patternFrequency = 6.0f;
7  public float seed = 94.5f;
```

Note that this includes a seed variable which allows different regions of the terrain space to be sampled. In this case, the seed variable does correspond to a position in terrain space so can be animated to provide an infinite terrain scroller.

```

1  void Update () {
2      On_UpdateMesh ();
3      seed += 0.01f;
4  }
```

6. A further point not discussed in much detail is the choice of (u, v) coordinates for each vertex in the terrain. The u value is chosen from the variation offered

by a noise function using only the horizontal coordinates of the vertex. However the v vertex is derived from the height of the terrain at that point. If a vertical gradient texture is used then the colour assigned to the terrain would depend on its height. This would allow different colours for grassy lowlands, rocky mid-level regions and snowy mountain tops.

A snapshot of the terrain shown in Figure 9.3.3, using only a generic image texture, shows distinct colour regions linked to height.

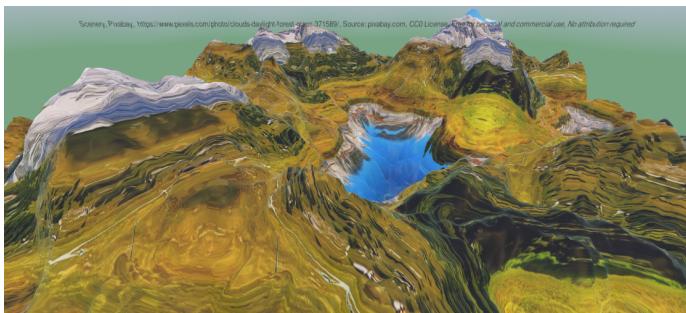


Figure 9.3.3: Terrain produced using procedural synthesis and with a height related texture mapping process.

10

Shades of Grey

10.1 Lighting and Texturing

10.1.1 Description

The appearance of objects in the virtual world is governed by a set of shaders managed by the virtual reality engine. These shaders are invoked each time the scene is redrawn to combine properties of the surface materials (e.g. colour, texture) with those of the lights in the scene (e.g. position, brightness) to determine the exact colour shade used to fill in every pixel in the rendered images. Most virtual reality engines provide a range of shaders that mimic those found in 3D modelling packages which provide a reasonable approximation to the more significant lighting effects observed in physical realities. They might also include a collection of shaders covering a range of specific effects that are popular in games and virtual worlds such as glowing particles or flame effects.

Shaders are actually scripts that run on the processor(s) on the graphics card. Modern graphics cards contain hundreds of processing elements (cores) that all run in parallel to achieve the rendering speed needed for virtual reality applications. Shaders can thus be modified or rewritten to exploit the processing power on the graphics card to produce other visual effects, or even other processing intensive but highly parallelizable tasks

that may be required in specific virtual reality applications.

The scripting language for shaders uses the same elements previously described in section 3.1 on page 79. Key differences are that most shaders have predefined types representing positions and colours, consisting of 4 element vectors. The elements may be referred to as x , y , z and w , or as r , g , b and a . Additional arithmetic operators are also predefined for working on these types.

There are two primary types of shader that are commonly used when rendering virtual worlds.

Vertex shaders receive a single vertex that may be any one of the vertices in the 3D meshes being rendered, and are responsible for transforming its position so that it is correctly placed relative to all other vertices in the world, and in the position in the viewing window corresponding to how it would look viewed from the camera (head mounted display position in virtual reality applications). Projection operations, including the stereoscopic projection used for each eye in a head mounted display, are also performed in this shader. Details of the transformation pipeline implemented in this process are described in section 2.6 on page 47.

Pixel (fragment) shaders receive the coordinates of a single pixel (or at least the fragment of a geometric surface covering a pixel) and are responsible for returning the colour of that pixel. The pixel shader has access to other properties of the vertex (or values interpolated from neighbouring vertices) such as the texture coordinates of the fragment so that texture information can be incorporated into determining the pixel's colour.

The common operations of lighting and texturing found in most standard shaders conform to established patterns. The principles behind these patterns are described here to help you identify where and how you might be able to modify these processes to achieve particular effects within your virtual worlds. Understanding this process can also provide insight into how to use

lighting and texturing effectively within a virtual environment, and to anticipate effects resulting from the limitations in the models used.

10.1.2 Pattern

Attempts to replicate lighting in computer generated imagery range from simple models (such as no-lighting, or constant colour), to approximations that take into account relative positioning of light and surface, all the way through to lookup tables using data recorded from scenes in physical reality. Partly due to tradition and partly for efficiency common shaders in virtual reality engines tend to model three major lighting effects: ambient, diffuse and specular reflection.

Ambient reflection assumes a single brightness for a surface, affected only by the colour of the surface and a global level of illumination originating from nowhere in particular. The surface colour is determined from a material property of the surface (assume this colour variable is named k_a) and the illumination level (I_a) is a global property which may be a setting in the engine, or a property of a specific world properties node in the scene graph.

The ambient illumination pattern is present when this expression pattern is present:

$$\text{pixel colour} = k_a I_a$$

Ambient lighting tends to show only a single colour for the entire surface which achieves the benefit of at least making the object visible but does not reveal any detail of the object's structure or motion.

Diffuse illumination is required to provide lighting effects that respond to the shape of the object. The brightness of the surface varies in proportion to the angle between the surface normal and direction towards the light source. Surfaces facing the light receive the most illumination, which drops off in proportion to the co-

sine of the angle until the surface normal is perpendicular to the rays from the light source. Surfaces that face away from the light source have their illumination levels clamped to 0.

A diffuse lighting model requires information from a number of different sources. The surface colour or texture, k_d , is a property of the object's material. The vertex information provides, after the transformation process in the vertex shader, the position of the vertex as well as the direction of the surface at that vertex. Direction is normally represented using a normal vector, N ; a vector oriented outwards perpendicular to the surface at that point.

Each of the light sources provides an additive contribution to the lighting. Each light source is an element in the scene graph with properties that include a light intensity, I_L , and a position from which a direction vector, L , pointing from vertex to light source can be calculated.

The diffuse illumination pattern takes the form:

$$\text{pixel colour} = k_d I_L \max(0, N \cdot L)$$

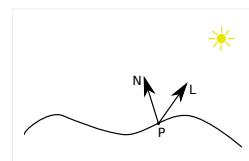


Figure 10.1.1: Diffuse lighting combines the geometry of the surface (direction indicated by the normal vector, N) with the direction towards the light source, L , calculated by subtracting the surface position from the light's position.

Colour values are added for each of the diffuse light sources and for other aspects of the light model.

The term $N \cdot L$ is equivalent to the cosine of the angle between N and L , provided both of the two vectors have unit length (have been normalized). The max expression is used to clamp intensity values to zero when the light is behind the surface since this would otherwise reflect negative amounts of light.

Specular illumination models shiny surfaces where a reflected image of the light source is visible in the surface. This reflected direction is just the vector from the surface to the light source, L , reflected about the vector perpendicular to the surface, N , and is represented with the value R . Many shader environments have a function (probably called "reflect") to calculate R from N and L , but it can also be generated from the expression:

$$R = 2(N \cdot L)N - L$$

Many shiny surfaces are not perfect reflectors but instead provide a somewhat blurred image of the light source known as a specular highlight. The degree of blur is controlled by a material property such as shininess, n . Since the position of the specular highlight is dependent on the viewing angle, we also need an additional vector from surface to the camera's position referred to as V .

The specular illumination pattern consists of expressions of the form:

$$\text{pixel colour} = k_s I_L (R \cdot V)^n$$

There are a range of specular reflectance models (this one is attributed to Phong [Phong, 1975]) all of which approximate the appearance of the specular highlights in slightly different ways.

Image textures can also be applied to the surface to provide the base material colour (k_a , k_d or k_s). Texture access makes use of texture coordinates, attributes of the vertex which are passed into the vertex and pixel shaders. These texture coordinates are normally named (u, v) or (s, t) . They represent the coordinates in the 2D image for which the colour of that point on the surface must be retrieved. Texture coordinates are often created through processes such as texture unwrapping in the 3D modelling package used to construct the geometric representation. They can also be procedurally generated as well based on other properties of the vertex or fragment. Procedurally generated texture coordinates are useful for adapting the way the texture is used dynamically to properties of the scene, such as for producing shadows.

The texture image itself is provided to the shader as one of the uniform ("global") parameters to the shader. It may frequently (pun intended) be referred to as a *sampler*. The name is appropriate in several ways: retrieving a value from the texture involves sampling. This may involve filters that aggregate regions of the texture if the sampled points are far apart in the texture space, or that perform interpolation between neighbouring values if the

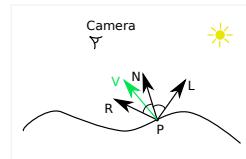


Figure 10.1.2: The specular lighting model considers how much of the light reflected from a shiny surface along the reflected vector, R , is sent in the direction of the camera (viewer) along vector V .

sampled points are less than a texel apart. Furthermore, remember that the texture image itself is just a lookup function; a sampled version of some potentially continuous function such as the scene which was photographed to provide an image texture. Hence the texture access process uses the same conventions as invoking a function using the texture coordinates.

The texture access pattern is as follows:

```

Sampler textureSampler : uniform variable
    passed into the shader
TexCoord (s, t) : property of the vertex
    provided to the graphics pipeline
// optional modification of (s, t)
texture value = textureSampler (s, t)
// optional use of texture value to set
// surface colour
colour = texture value

```

The colour produced from this process is used to set or modify the material colour of the surface (k_a , k_s or k_t) used in the lighting models, to combine lighting with a texture pattern across the surface.

Interesting texturing opportunities arise when considering that: textures don't have to only be 2D. They can be 1D, but 3D and 4D are usually supported. The contents of a texture don't have to be colour values, but can be other information useful for other terms in the lighting expressions but also as parameters for other surface appearance processes.

10.1.3 Example

Example 41. Applying the pattern to Unity software

1. This example creates a custom shader for Unity software that provides a form of non-photorealistic rendering. In particular it emulates a cartoon (toon) shading approach that uses three colours for regions that

are mostly: unlit, lit with just diffuse illumination, and lit with a specular highlight.

Start with a new scene and add an object with some non-trivial surface geometry (i.e. not a cube, sphere or cylinder). A capsule primitive is reasonable, but consider also including some more interesting objects as well.

2. In the project assets, create a new shader of the category: Unlit Shader, and name it ToonShader. Create a new material to make use of this shader. In the inspector pane, set the shader property to use the ToonShader (probably under the Unlit options). Apply this material to any objects that you want to have the toon shading effect.

Double click on the shader to open it in the Editor.

3. The fragment (pixel) shader is responsible for providing the colour to any individual pixel. We'll start off by including a standard diffuse lighting calculation here, before quantizing this to produce some toon effects.

The diffuse lighting calculation requires several elements:

- (a) The normal vector at that pixel. We need Unity software to pass that through from the application, and via the vertex shader to the fragment shader. Add the normal entry to the appdata structure. This ensures that Unity software includes this information as an additional property for each vertex.

```

1 struct appdata
2 {
3     float4 vertex : POSITION;
4     float2 uv : TEXCOORD0;
5     float3 normal : NORMAL;
6 };

```

The next step is to pass the normal vector through the vertex shader. This involves transforming the

vector in the vertex shader (section 2.6), and assigning the result to a new field in the v2f structure (which passes information from vertex **to** fragment shaders).

The transformation added to the vertex shader is achieved by the line:

```
1 o.nor = normalize (mul ((float3x3)
    UNITY_MATRIX_IT_MV, v.normal));
```

The extra field in the v2f structure results in the following structure:

```
1 struct v2f
2 {
3     float2 uv : TEXCOORD0;
4     UNITY_FOG_COORDS(1)
5     float4 vertex : SV_POSITION;
6     float3 nor : NORMAL;
7 };
```

- (b) The light direction is assumed to be constant for all pixels in the scene as if the light were an infinite distance away. This is provided to the shader as a property through the Unity software editor.

```
1 LightDirection ("LightDirection", Vector) =
(0,1,1,0)
```

This then also needs to be declared in the body of the shader as well, with the other property variables just above the vertex shader.

```
1 float4 LightDirection;
```

- (c) The diffuse lighting coefficient, k_d . This is also constant across the surface being shaded so, like the light direction, is provided by a value editable from the Unity software inspector. As before, in the properties section of the shader include:

```
1 kd ("kd", vector) = (1, 0, 0, 1)
```

Declare the variable along with the others just above the vertex shader.

```
1 float4 kd;
```

- (d) The light intensity, I_L , is assumed to be 1 in this case so this term is ignored.

The diffuse lighting calculation then becomes:

```

1 float IL = 1.0;
2 float3 N = i.nor;
3 float4 L = LightDirection;
4 col = kd * IL * max (0, dot (N, L));

```

Use this at the end of the fragment shader to set the colour returned. The resulting shape should then receive a surface appearance as shown in Figure 10.1.3.

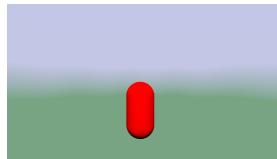


Figure 10.1.3: Diffuse shading applied through the use of the fragment shader.

4. This is converted into a toon-like mode by restricting it according to the value of the dot product. Low values of the dot product produce a single darker version of the colour in k_d , while high values produce a single brighter shade. An intermediate region is coloured black if desired to emulate the pen boundaries that might be drawn by hand-drawn cartoons.

Replace the diffuse lighting calculation with the toon shader variation, which produce output such as shown in Figure 10.1.4.

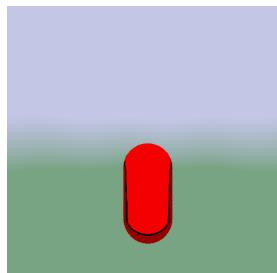


Figure 10.1.4: Toon shading by restricting the shades that are produced by the diffuse lighting model.

```

1  float diffuseDot = max (0, dot (N, L));
2  if (diffuseDot < 0.5)
3  {
4      col = kd * 0.3;
5  }
6  else
7  {
8      if (diffuseDot < 0.6)
9      {
10         col = kd * 0.0;
11     }
12     else
13     {
14         col = kd * 0.9;
15     }
16 }
```

5. The specular highlight is included by adding in the specular lighting model. The parameters used for this are k_s , the specular reflectivity (colour) and n , the lighting exponent which is referred to using the name “gloss” for clarity.

These are defined as part of the properties available to the shader via the Unity software editor.

```

1  ks ("ks", vector) = (0, 1, 0, 1)
2  gloss ("gloss", float) = 15.0
```

They also need to be declared as variables in the shader, just above the vertex shader.

```

1  float4 ks;
2  float gloss;
```

The specular model is implemented using the expression described. The other required values are the view vector, V , which is easily determined as the camera is always facing down the z-axis in the view/screen coordinate system used in the fragment shader. The reflection vector, R , is derived from the L and N vectors. The shading language has a specific function to compute it. Include the code below after the diffuse computation.

```

1  float3 V = float3 (0, 0, -1); // view direction is
   constant in view coordinates.
```

```

2 float3 R = reflect (L, N);
3 col = col + float4 (ks * IL * pow (max (0, dot (V,
R)), gloss));

```

The outcome of including specular effects is shown in Figure 10.1.5.

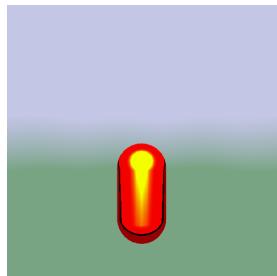


Figure 10.1.5: Specular effects overlapping with the diffuse toon shader.

6. The cartoon version of the specular effects are implemented by using threshold levels of the specular expression. Replacing the specular computation with this version yields the appearance shown in Figure 10.1.6.

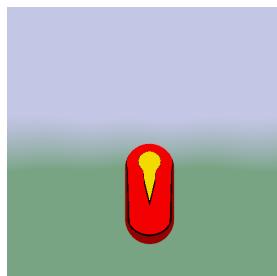


Figure 10.1.6: Toon shader supporting both diffuse and specular effects.

```

1 float specularDot = pow (max (0, dot (V, R)), gloss
    );
2 if (specularDot < 0.6)
3 {
4     // no specular contribution, just diffuse layer.
5 }
6 else
7 {
8     if (specularDot < 0.8)
9     {
10         col = col * 0.0; // black border
11     }

```

```

12    else
13    {
14        col = col + ks * 0.7;
15    }
16 }
```

7. The exposed properties of the shader are manipulated through the material component within the inspector panel. Unlike normal scripts, shaders are run even during the editing phase of the project so changes to the visual output are immediate.

10.2 Bump mapping

10.2.1 Description

Creating or modifying shaders provides opportunities to incorporate custom visual effects into your virtual reality applications. These may be as trivial as just toggling a default setting that cannot be modified in any other way (such as producing two sided materials in one well known engine), to including particular lighting and material effects that are specific to a significant class of the objects in your application.

Bump mapping is a facility that is already supported in some form in many engines. It does, however, provide an excellent example of the interaction of lighting and texturing in less conventional ways to achieve dramatic effects. The form of bump mapping described here usually falls under the title of normal mapping.

Key points about the process involved:

- The only parameter of the lighting calculation that really depends on the shape of the object is the normal vector, N . Other dependencies (such as the light vector L) are not critical unless the light happens to be really close to the surface. Thus if we want to add lots of surface detail to objects we can either add lots more vertices (hard work, but see displacement mapping), or we can modify the normal vector after it gets

interpolated to each pixel, but before it gets used in the lighting calculation. The effect of bump/normal mapping is then to shade the surface to look like it has more detail although it has really not been physically altered.

- The changes to the normal vector are supplied by a texture containing details of how the direction of this vector must change. This texture can be procedurally generated or created through some other process, but must provide a direction vector at each texel rather than a colour value. An image search for normal maps usually returns lots of blue-tinted pictures. This is because the x, y and z-directions of the vectors stored at each texel are being shown as colours (matching x to red, y to green and z to blue). Most normal maps try to make the surface appear to face outwards (the local z direction) and so have a dominant blue shade.
- The modification to the surface direction must be relative to the current surface orientation at each vertex. Otherwise the apparent deformation would be in different directions depending on whether it occurs at the front, back or side of the object. We need to transform the vector in the normal texture into the local coordinate frame at each vertex before modifying the normal vector with it. The local coordinate frame at a vertex assumes that the current normal vector is facing outwards so this becomes the z direction. The x direction depends on the tangent vector at the vertex (this is a vector parallel to the surface which may need to be provided as a property of each vertex by the 3D modelling package). The y direction of the local coordinate frame is just a vector perpendicular to both x and z directions, also referred to as a binormal.

10.2.2 Pattern

The pattern for a bump mapping shader is:

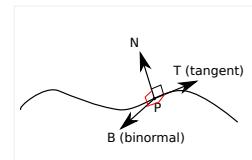


Figure 10.2.1: Bump and normal mapping consider how the normal vector would be modified by a value retrieved from a texture. This change is relative to the local coordinate system at any point of the surface defined by the original normal vector, a tangent vector and the binormal direction vector.

```

    Sampler normalMapSampler

    function vertexShader (Vertex v)
    {
        transform v.position
        transform v.normal
        transform v.tangent
        v.binormal = v.normal × v.tangent
    }

    // rasterizer stage automatically interpolates
    // vertex properties to each pixel
    function pixelShader (Pixel p)
    {

        ΔN = 2 * normalMapSampler (p.s, p.t) - 1
        Matrix tangentSpaceTransform =
            Matrix (p.tangent, p.binormal, p.normal)
        N = tangentSpaceTransform * ΔN
        do lighting calculation using N
    }
}

```

The values in the texture provided are often retrieved by the default colour texture file loading process. Colour values are clipped to the range $[0, 1]$ whereas the values required for a unit length normal vector are either both positive and negative in the range $[-1, 1]$. Normal map values are scaled and offset before being stored as images, so we need to reverse the process when retrieving values from image textures containing normal maps. The expression: $n = 2c - 1$ transforms a value c in the range $[0, 1]$ to a value n in the range $[-1, 1]$. When applied to a colour or vector, it applies this process to each (r,g,b or x,y,z) component.

A useful property of matrices allows us to achieve conversion of a vector into the local coordinate frame by simply creating a matrix whose columns are the 3

axis directions of the local coordinate frame, and just multiplying the vector by this matrix.

Displacement mapping is achieved by retrieving and transforming the normal vector in the vertex shader, and using this value to update the position of the vertex ($v.\text{position} = v.\text{position} + N$). This does modify the silhouette of the object, but does require a very detailed mesh (lots of vertices) to accurately portray the fine structures in the normal map. Oculus also recommends parallax mapping as a better way of adding texture detail to objects than normal mapping¹.

10.2.3 Example

Example 42. Applying the pattern to Unity software

1. This example illustrates the use of textures for purposes other than adding colour to the surface of an object. In this particular case we modify the properties of the lighting calculation to make it appear as if the surface had more detail; detail that is contributed by a texture image containing changes to the surface normal vector.
2. The first stage is to create a new scene, add an object, create a custom material for this object, and apply that material to the object. Find a normal map texture image (an internet search usually yields several of these), and get the material to use this. Initially add the image as a texture to the Albedo channel in the Inspector pane so the image sets the colour of the surface (we'll remove this later). The initial set up is shown in Figure 10.2.2.
3. The next step involves creating a custom shader to access the texture information and produce the same output. Create a new shader, of the Unlit type, and name this NormalMapShader. Set the material to use this shader, by changing the shader property of the material in the inspector pane. The new shader should

¹ <https://developer.oculus.com/design/latest/concepts/bp-rendering/>

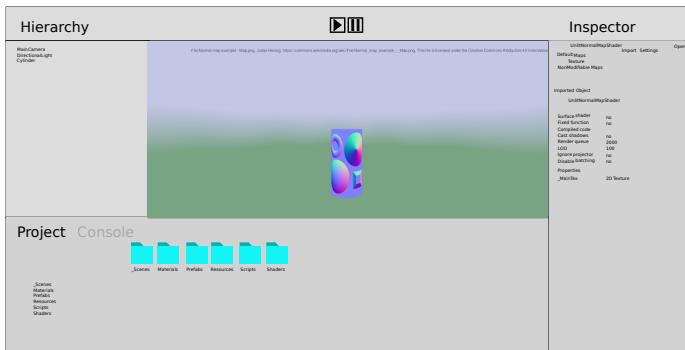


Figure 10.2.2: Normal map texture mapped to the colour channel of the surface of an object.

pick up the texture property but apply the normal map image again if it doesn't. Now edit the shader.

4. The texture mapping process in the fragment shader currently picks up the colour value from the texture and sets the colour of the fragment to this colour. The line that achieves this is:

```
1 fixed4 col = tex2D(_MainTex, i.uv);
```

We revise this to set the colour to a fixed value, and instead convert the value from the image to its intended purpose; a vector describing an update to the surface's normal vector. Replace the fragment shader with this code:

```
1 fixed4 frag (v2f i) : SV_Target
2 {
3     fixed4 col = float4 (0.6, 0.6, 0.6, 1.0);
4     float3 normalmap = 2 * tex2D(_MainTex,
5                                     TRANSFORM_TEX (i.uv, _MainTex)) - 1;
6 }
```

This sets the surface colour to a boring uniform grey. The normal vector extracted from the image has been compressed down into the range $[0, 1]$ used for colour values, rather than the range $[-1, 1]$ used by normalized vectors, and so has to be scaled back by using an expression of the form: $vector = 2 * colour - 1$ for each of the x, y and z (or r, g and b) components.

5. We need a lighting model implemented to achieve normal mapping, so use a diffuse lighting model. The changes required to achieve this are:

- (a) Enable the normal vector as part of the vertex attributes by adding the line below to the appdata structure:

```
1 float3 normal : NORMAL;
```

- (b) Transform the normal vector in the vertex shader using the line:

```
1 o.normal = normalize (mul ((float3x3)
    UNITY_MATRIX_IT_MV, v.normal));
```

- (c) Add the resulting transformed normal vector to the v2f structure so that it is communicated to the fragment shader.

```
1 float3 normal : NORMAL;
```

- (d) Update the fragment shader to include the Unity software lighting model support functions, and use a diffuse lighting model to control the surface colour

```
1 #include "UnityLightingCommon.cginc"
2
3 fixed4 frag (v2f i) : SV_Target
4 {
5     fixed4 col = float4 (0.6, 0.6, 0.6, 1.0);
6     float3 normalmap = 2 * tex2D (_MainTex,
        TRANSFORM_TEX (i.uv, _MainTex)) - 1;
7
8     half3 worldNormal = UnityObjectToWorldNormal (i
        .normal);
9     half nl = max(0, dot(worldNormal,
        _WorldSpaceLightPos0.xyz));
10    col = col * nl * _LightColor0;
11    return col;
12 }
```

The resulting object should display a smooth colour gradient emulating diffuse lighting.

6. The normal mapping process requires a local coordinate system at each pixel, consisting of the normal, tangent and binormal vectors. In a process similar to that used for propagating the normal vector for the lighting calculation, add the following elements to provide the other two vectors.

The appdata structure needs an additional vertex attribute:

```
1 float3 tangent: TANGENT;
```

The v2f structure needs to produce an additional two vectors. The binormal is calculated from the normal and tangent vectors and so is not required as a separate vertex attribute.

```
1 float3 tang: TANGENT;
2 float3 binor: FLOAT3;
```

The vertex shader needs to compute values for these additional fields.

```
1 o.tang = normalize (mul((float3x3)
    UNITY_MATRIX_IT_MV, v.tangent));
2 o.binor = normalize (cross( o.nor, o.tang));
```

7. The final step is to modify the surface normal vector to include the value extracted from the normal map.

```
1 fixed4 frag (v2f i) : SV_Target
2 {
3     fixed4 col = float4 (0.6, 0.6, 0.6, 1.0);
4     float3 normalmap = 2 * tex2D(_MainTex,
        TRANSFORM_TEX (i.uv, _MainTex)) - 1;
5
6     float3x3 tangentSpaceTransform = float3x3 (i.tang
        , i.binor, i.normal);
7     i.normal += normalize (mul (normalmap,
        tangentSpaceTransform));
8
9     half3 worldNormal = UnityObjectToWorldNormal (i.
        normal);
10    half nl = max(0, dot(worldNormal,
        _WorldSpaceLightPos0.xyz));
11    col = col * nl * _LightColor0;
12    return col;
13 }
```

8. The resulting surface effect is shown in Figure 10.2.3.

The surface colour is no longer controlled by the image texture as was initially the case. Instead the value from the image is converted into a modification to the surface normal vector prior to the use of this vector in the lighting calculation. Note that this shader should

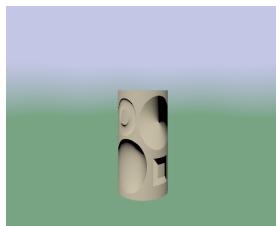


Figure 10.2.3: Normal mapping applied to the surface of a cylinder using a shader.

be using a camera space light position in the fragment shader, rather than the world space position value that is utilized.

10.3 Combining values

10.3.1 Description

Consider the situation where a surface can have two textures applied to it. For example, a scaly furry demon creature needs the scales texture in some areas, and the fur texture in others. We have some factor f which tells us how much of each texture is to be applied. A value of 0 for f means that we only apply the scales texture, and a value of 1 for f means that we only apply the fur texture. Values of f between 0 and 1 mean that a blend of the two textures is to be used, in proportion to the value f . So a value of $f = 0.2$ is closest to the scales value, so we would combine 80% scales with 20% fur.

The source of the value f could come from a third texture that has been painted over the object in a modelling package. Dark areas would correspond to low values of f (i.e. scales) and bright areas would be the fur region. Intermediate values are produced by smoothing the

boundary producing a gradual blend of one texture into the other.

This process could be applied to numerous other situations as well. Consider an object that is in either location A or location B . The extent is controlled by factor f where $f = 0$ corresponds to location A and $f = 1$ corresponds to location B . Our value f , perhaps controlled by a slider, can have values between 0 and 1 corresponding to the object being an appropriate distance between A and B .

10.3.2 Pattern

The process of linear interpolation involves two quantities, A and B . We need a mixture of these two, controlled by factor f such that:

- When $f = 0$, the value must be A .
- When $f = 1$, the value must be B .
- As f varies from 0 to 1, the value must change from A to B by reducing the proportion of A and increasing the proportion of B .

The linear interpolation pattern uses the expression:

$$(1 - f) A + fB$$

This is equivalent to regarding A and B as points on a line and finding the points on the line segment joining them. Coincidentally this expression is used for drawing lines and rays by incrementing f at regular intervals and plotting the resulting point.

The expression returns values when $f < 0$ or when $f > 1$. However interpolation operations may prefer to clip the output in these cases:

$$\begin{cases} A & \text{if } f < 0 \\ (1 - f) A + fB & \text{if } 0 \leq f \leq 1 \\ B & \text{if } f > 1 \end{cases}$$

10.3.3 Example

Example 43. Applying the pattern to Unity software

1. The interpolation pattern is useful for blending between different values. Typical examples of such values are colours, used to create a colour gradient, or positions, used to make an object between two defined end points.

The example used here creates a guard that patrols from one position to the another.

Create a Unity software scene. Equip it with two objects to mark the end points of movement, and a guard object to which a C# script named Patrol is added, as illustrated in Figure 10.3.1.



Figure 10.3.1: The interpolation pattern applied to an object moving between two defined endpoints.

2. The interpolation pattern is implemented in the script shown in Algorithm 87. Remember to set the positions of the end points in the two public variables by dragging the end point objects into the two properties of the Patrol component in the Unity software inspector pane. The interpolation parameter, t , needs to vary between 0 and 1. Currently this is just set from the current time which makes constraining the range and speed of motion awkward.

The forward direction is set from the tangent vector to this expression, achieved by differentiating the interpolation formula. Ah, the value of learning calculus.

Algorithm 87 The interpolation pattern applied.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Patrol : MonoBehaviour {
6
7      public Transform startPoint;
8      public Transform endPoint;
9
10     void Update () {
11         float time = Time.time;
12         float t = time;
13
14         transform.position = (1.0f - t) * startPoint.
15             position + (t) * endPoint.position;
16         transform.forward = -1.0f * startPoint.position +
17             1.0f * endPoint.position;
18     }
19 }
```

3. The issue with range and speed is addressed by using a periodic function (section 9.2 on page 299) to modulate time.

```

1  float time = Time.time;
2  float frequency = 0.2f;
3  float t = 0.5f + 0.5f * Mathf.Sin (frequency * time
   );
4
5  transform.position = (1.0f - t) * startPoint.
   position + (t) * endPoint.position;
6  transform.forward = (-1.0f * startPoint.position +
   1.0f * endPoint.position) * Mathf.Cos (
   frequency * time);
```

A Sin function converts an ever increasing time value into a value that oscillates continuously between -1 and $+1$. Since the interpolation parameter, t , needs to be in the range $[0, 1]$, we convert using $t = \frac{1}{2} + \frac{1}{2} \sin(\text{time})$. A frequency term allows the rate of oscillation to be controlled. As a side effect of the smooth sinusoidal function, the patrolling guard slows down as it approaches each turning point.

The forward vector still uses the derivative of the in-

terpolation expression $\frac{dt}{dtime}$, but the astute reader will note some missing terms. Since these are purely for scaling and the forward vector is normalized (forced to unit length), we can leave these elements out.

Figure 10.3.2 shows a frame from the patrolling process.



Figure 10.3.2: Object patrolling between two end points.

11

Finishing Touches

11.1 Getting started

11.1.1 Description

The end of a book on developing virtual reality applications might seem like an odd place to describe getting starting on populating virtual worlds. This section deals primarily with the manual processes for setting up and managing projects. This is not fundamental to the procedural reasoning processes that are used in developing virtual reality applications but does introduce some philosophy around sanitary habits while working on such applications. Ideally you're skimmed ahead, or been tipped off by a wily teacher and used this section for providing an introduction to the virtual reality engine that you're working with.

Virtual reality engines use the concept of a project to refer to all the assets (including program code) that are associated with a particular application. This needs to be managed in a systematic fashion as any reasonable sized application soon gets unwieldy. Conventions around the naming of your various assets should be adhered to for much the same reasoning as we adopt these conventions when developing program code. Following an accepted convention means that other team members can work with, and contribute to the project, without hav-

ing to decipher any personal conventions that you might otherwise adopt. Maintaining and growing the project just builds on a standard pattern without the need to rearrange everything once the project gets beyond a particular size.

A project supports many scenes. Each scene is roughly equivalent to its own virtual world. A typical application might have one or more scenes to represent the core virtual world required by the application, but may have others representing introduction, title, instruction, or credit scenes. There is no obligation to have all of these scenes, but breaking up the world into smaller sections with portals between them can reduce the complexity of any single scene and help with maintaining performance levels. There is a delay in switching scenes which detracts from the experience if the user does not expect it. Scenes for testing particular elements of the virtual world can also be created so that experimentation does not endanger any production scenes.

The structure of a scene often depends on the conventions of the organization developing the virtual reality application. While the philosophy espoused throughout this book is to manage as much as possible procedurally, there are some elements that are once off content or that need significant manual manipulation that would not be effective to automate. These elements need to be added to the scene by hand in a way that still allows the structure of the scene graph to be easily managed and manipulated. The parent-child relationships in the scene graph are used to group related elements together under a single (collapsible) heading produced using an empty/invisible scene element. Since the parent-child hierarchy affects the ways in which objects are transformed (section 2.6 on page 47) care should be taken when objects under different sections need to interact, share transformations or be attached to one another.

11.1.2 Pattern

A typical project structure is normally broken down into a number of standardized containers or folders. These should be named for what they are. Each of these folders should then have subfolders so that the various subcategories are clearly distinguished. As a suggestion, no folder should have more than 10 elements in it unless all elements are clearly just anonymous parts of a single element (e.g. all 100 frames for an animation sequence could probably live in a single subfolder, assuming they needed to be stored as separate objects).

The collection of scenes is probably the most accessed folder and relevant when first starting to work on an existing project. Placing it as the first folder in the project provides convenient access. There are tricks that can be played by pre-pending particular symbols to the folder name to force it to the top when sorted alphabetically.

A typical top level folder structure pattern would be:

- Scenes: the folder containing all the scenes in the project.
- Prefabs/Templates: templates for scene elements that are procedurally instantiated.
- Scripts: contains the program files for any procedural elements used in the application.
- Models: any 3D geometry defined for objects in the scene.
- Resources: any other assets produced by external applications that are required. Since these are generally not modifiable within the virtual reality engine, this folder acts a repository for these assets, and as a standard location to refer to them. This often contains sub-folders for Textures and Animation sequences.
- Materials: any custom surface materials applied to particular objects.

- Shaders: surface shader code. This could also be a sub-folder under scripts but, since a different programming language is involved, is often kept as a separate top-level folder.

New scenes use a similar convention since they contain a number of standard elements. A suggested pattern would be:

- Camera: cameras are often provided by default, as the scene cannot be viewed without one. Usually any virtual reality system has its own tracked camera object, which needs to override the default element. A default camera is useful when developing so testing can take place without donning the head mounted display each time.
- Lights: omitting lights from the scene results in a completely black image (usually). The static lights in the scene can usually be grouped under a common parent. Moving lights (such as a torch carried by the user) would need to be handled separately.
- Static scene elements: Any geometric elements of the scene that do not move can be grouped under a common parent.
- Avatar: the representation of the user, and its associations with various display and controller peripherals, can often be collected together under a single top-level parent.
- Terrain: while terrain is a static scene element there may be a case for providing this as a separate scene element. Terrain can involve various engine manipulable content, such as a height field, which would be worth managing separately from those elements with fixed geometry.
- Managers: any managers in the scene could be grouped under this parent object. Some managers

adopt the managed objects as children, so the transformation of the manager object, and of the parent managers object may need to be handled carefully.

- Instantiated objects: Rather than a single category, it is worth having a top-level element for each class of instantiated object. Searching through elements in the scene graph is faster if only a particular portion of the graph need be searched (i.e. children of a particular parent). This also allows a potentially long and dynamic list shown in the scene graph inspector to be collapsed to a single unchanging element which helps with visual inspection of the scene graph when debugging.
- Standard Assets: Sometimes we use third parties script elements or plugins. It is convenient to bundle these under this folder to avoid getting them mixed up with your organization's content. This way, if the plugin is updated at some point in the future, we know we only have to replace the material in this folder.

Top level parent objects in the scene graph are ideally placed at the origin aligned to the coordinate axes. This avoids having to compensate for the parent object's transformation when manipulating any of their child nodes.

11.1.3 Example

Example 44. Applying the pattern to Unity software

1. The following steps represent a quick-start guide to creating a new virtual reality application project within Unity software. Once Unity software has started up, it provides options to open an existing project, or create a new one. We are creating a new project. The location of the project should be chosen as a work area with plenty of storage space, and one that permits reasonably fast file access as the development process involves frequent accesses to files, including

checking and compiling scripts. The name of the project determines the folder within this area that is used to store the project files.

Remember both location and project name as this is the folder that contains all the files that are required if you need to backup your project or copy it to another machine. You can also directly access files within this folder in case you need to directly update asset files managed by other applications (such as 3D models or textures).

Enter location and project name, and create a new project.

2. The startup screen appears looking similar to that shown in Figure 11.1.1. Layout of the individual panes, and other elements may vary according to your version and configuration. Key elements to look out for are: the scene graph mostly represented in the Hierarchy window, with details of components and properties in the Inspector pane. The Project pane is used to structure the project assets. A view of the scene through the camera is in the Game pane, while arbitrary viewpoints are selected by switching that pane to the Scene tab. The Console tab on the Project pane shows any errors that have been identified in the scene, or any scripts, and can also report additional information that you might need for debugging.

Of the controls at the top of the window, the play button is used to start the virtual reality application. It is toggled once it turns blue to stop the application. While changes can be made while the application is running, they only last if they are made when the application is stopped. For emphasis: **do not make changes to your application while it is running**. In practice, changing properties while the application is running is a handy way of testing out the effect of changes without having to worry about resetting everything back to defaults afterwards (Unity software

does that for you).

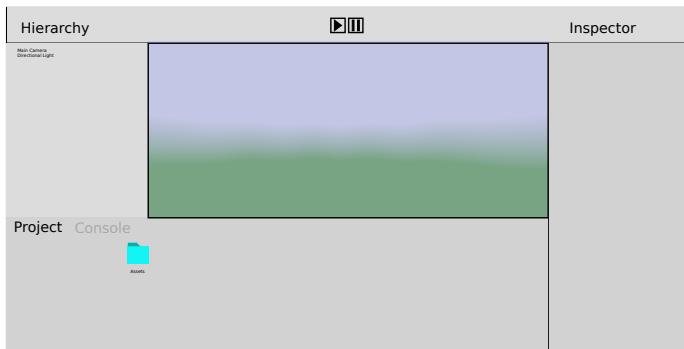


Figure 11.1.1: The Unity software interface after creating a new project.

3. New scenes are automatically equipped with a camera and light. The camera is positioned 10 units down the negative z-axis to make it easier to see new objects which are usually added at the origin. Click on any of the objects in the Hierarchy pane to see its properties in the Inspector window.
4. We'll start by creating the standard folders for our assets. In the project folder, right click on the Assets window and select Create/Folder. Press F2 to rename the folder, and name it "_Scenes". The _ makes sure it stays first in the list. Repeat this process for the other standard folders shown in Figure 11.1.2. It is a good idea to do this for every project unless it is definitely a small throwaway project. Enter any of the folders to

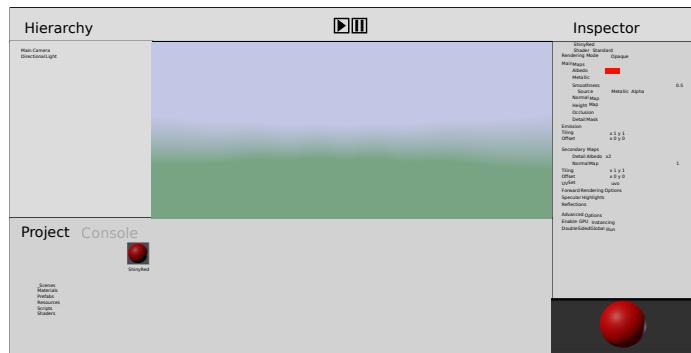


Figure 11.1.2: A typical standard structure for project asset folders.

nominate them as the current folder for adding new assets.

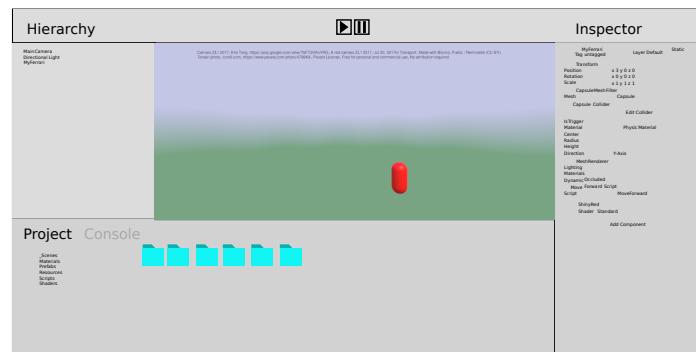
5. As an example of asset creation we create a new material in the Materials folder. Select the Materials folder, and right-click the Create/Material. Use F2 to rename to ShinyRed. Click on the white box next to Albedo in the Inspector pane, and select a reddish colour from the colour picker that pops up (Figure 11.1.3). We now have a custom material asset that we can apply to scene elements.

Figure 11.1.3: New assets creation illustrated through the process of creating a new material asset.



6. The next step is to demonstrate creating new elements in the scene. Click on the Create button just below the label of the Hierarchy window, and add a 3D Object/-Capsule to the scene. Again, press F2 and replace the default name "Capsule" with a more relevant name for the object, say "MyFerrari". A vehicle like that deserves an appropriate colour, so drag the material over from the project window and drop it on the "MyFerrari" label in the Hierarchy window (Figure 11.1.4).
7. We'll add another asset to the capsule car; a script element to make it move. In the Project pane, under scripts, right-click and Create/C# Script. Rename this to "MoveForward". Some parts of the script are automatically generated for you, and are visible in the Inspector Pane. We'll need to edit this using the development environment. Unity software supports several development environments, selectable under the Edit/Preferences/External Tools/External Script

Figure 11.1.4: New scene elements are added to the scene graph in the hierarchy window, and assets from the project folder are applied to them.



Editor option. MonoDevelop is suggested.

Double click on the script asset to open it in the code development environment. Add lines of code as shown in Algorithm 88. Once you've saved the file in the development environment, check the console tab on the Assets pane. If there are any red error messages then check carefully that you've entered the code correctly. Punctuation, spaces and upper/lower case letter can all be significant. Now drag this code asset onto our red car capsule object. A new MoveForward component should have been added to the object as shown in the Inspector pane. Press the play button (top of the Unity software window) and watch your car drive off into the distance. You should also be able to see the position property of the car change in the Inspector pane as it does this (Figure 11.1.5).

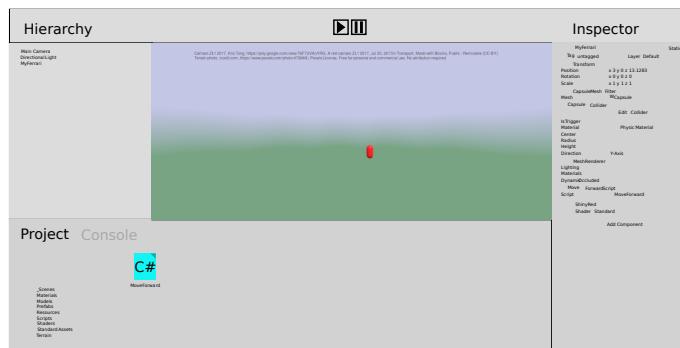


Figure 11.1.5: Running the application once the code asset is added as a component of the car object.

Algorithm 88 A script to move an object. This applies to the object it is attached to. The code consists of an extra line in the Update function, and potentially a new name for the class which must match what you called your asset.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MoveForward : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9
10     }
11
12     // Update is called once per frame
13     void Update () {
14         transform.position = transform.position + new
15             Vector3 (0, 0, 1) * Time.deltaTime;
16     }
}
```

8. Now that we have perfected the actions of our car, we convert it into a Prefab so that it becomes the template for a whole fleet of cars within our application. Drag the “MyFerrari” object from the Hierarchy window into the Prefabs folder of the Project window. The entry in the Hierarchy window turns blue, indicating that the object is a prefab - a template and warning you that modifying the instance in the Hierarchy view may not update the master version now part of the Assets.

Select the blue “MyFerrari” in the Hierarchy view, and press delete. It is now removed from the scene, but still persists as a template in the Prefabs folder in the Assets window.

We can now instantiate as many as well like. Drag it over 3 times from the Prefabs folder and drop it in the Hierarchy pane. That should create three capsules in the scene, all with slightly different names. Since

they're all right on top of one another, we may only see one. Select each in turn, and edit their position properties to put them at different locations (e.g. at -2,0,0; 0,0,0; and 2 1 0). Now press play and watch the entire cavalcade, each with the colour and motion script defined by the prefab, drive off.

9. To make the scene look more realistic we can head off to the asset store (the tab next to the Game view) and search for some freely available car models. This requires a Unity account which you may need to create for yourself. You can download and import the package and it is added to your assets folder. It is recommended you move this to your Standard Assets folder to avoid getting it mixed with your own assets, and so you can clearly identify 3rd party resources at later stages. Models can also be retrieved from other sources, such those you develop in 3D modelling packages, or download from many of the sites offering free assets.

To use this asset, follow the same process as described in the previous step and make a new prefab. Drag the asset into the scene, add the script component to it, drag it back to the prefabs folder, and delete the original in the scene. Now add several of the new prefabs to the scene again. You can also remove the capsule cars from the scene if they no longer look cool enough.

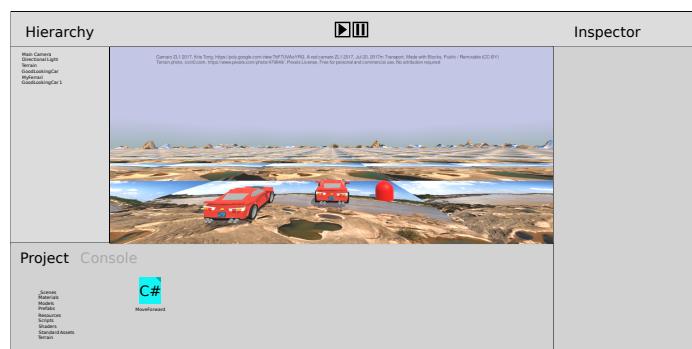
10. As a further cosmetic tweak we add terrain to the scene so we don't feel like we are floating in space. This provides a reassuring platform to explore our virtual world without having to wonder about the dangers of falling forever. In the Hierarchy view, use the Create button to add 3D Object/Terrain. This also creates a "New Terrain" element in the Assets window, which we know should be given a meaningful name and placed in a Terrain folder (Figure 11.1.6).

The terrain is placed with its corner on the origin.

To center it about the origin, set the position in the Inspector to -250, 0, -250 (the width and height of terrain are 500 by default). The terrain component in the Inspector has several controls for editing terrain. Some of the useful ones:

- (a) Download a grassy texture and add it to the Resources/Textures folder. If you remember where your project is located, you can copy the image file directly into that folder. In the Terrain component Inspector, select the Paint Texture option, and add the texture (under Edit Textures). Experiment with adding multiple textures and blending between them with the paint tools.
- (b) Use the Raise/Lower Terrain, or Paint Height to paint bumps onto the terrain.
- (c) Use the Place Trees option to paint trees onto the terrain. If your Unity software installation has the standard assets installed, the menu option: Assets/Import Package/Terrain Assets allows you to add tree models to your project.

Figure 11.1.6: The project with some cosmetic enhancements in the form of externally produced car models, and a terrain object with a texture image added.



12

Resources

Software

A repository with the project files for all examples is available at: <https://github.com/incshaun/MonsterFun>.

Credits

The following resources are used in this document, and in the accompanying software archive. The generosity of the creators of this content is gratefully acknowledged.

- Pineapple, Poly by Google, https://poly.google.com/view/7B_hERcgTZW, fruit, Oct 11, 2017, Food & Drink, Uploaded 3D Model, Public / Remixable (CC-BY)
- French fries, Poly by Google, <https://poly.google.com/view/4hpYt9IW-1k>, #potato #fastfood #chips, Oct 23, 2017, Food & Drink, Uploaded 3D Model, Public / Remixable (CC-BY)
- Green Pepper, Poly by Google, <https://poly.google.com/view/5d6zv40w2t0>, #pepper #vegetable #green-pepper #cooking, Oct 5, 2017, Food & Drink, Uploaded 3D Model, Public / Remixable (CC-BY)
- Brown Coconut, Cheryl Fong, <https://poly.google.com/view/cFAbF9p03-N>, A brown Coconut., Jul 22,

2017, Nature, Made with Blocks, Public / Remixable (CC-BY)

- Unicorn, Poly by Google, <https://poly.google.com/view/212RNECqFCA>, #majestic #magical #fairytales #rare #mystical, Sep 28, 2017, Animals & Pets, Uploaded 3D Model, Public / Remixable (CC-BY)
- Flying saucer, Poly by Google, https://poly.google.com/view/fojR5i3h_nh, #ufo #space #aliens #space-ship, Oct 24, 2017, Transport, Uploaded 3D Model, Public / Remixable (CC-BY)
- Rock, Poly by Google, <https://poly.google.com/view/dmRuyy1VXEv>, #stone #boulder #cliff, Oct 23, 2017, Nature, Uploaded 3D Model, Public / Remixable (CC-BY)
- Gavel image, Pixabay, Source: pixabay.com, <https://www.pexels.com/photo/534204/>, CCo License, Free for personal and commercial use, No attribution required.
- Wood chair, Poly by Google, <https://poly.google.com/view/7Jl72KgiRl->, #seat #rest #furniture, Oct 23, 2017, Objects, Uploaded 3D Model, Public / Remixable (CC-BY)
- Table, Poly by Google, <https://poly.google.com/view/8cnrw1Awqx7>, #furniture #dining #surface, Oct 23, 2017, Objects, Uploaded 3D Model, Public / Remixable (CC-BY)
- Television, Poly by Google, <https://poly.google.com/view/6Fa29JN3A3F>, #electronic #watch #screen #tv, Oct 24, 2017, Tools & Technology, Uploaded 3D Model, Public / Remixable (CC-BY)
- Birdhouse for chickadees, Sieuwe Maas, <https://poly.google.com/view/5-NoTPDZ0WZ>, A simple birdhouse for chickadees, Jan 6, 2018, Objects, Uploaded 3D Model, Public / Remixable (CC-BY)

- English: (oo2) WIKIPEDIA ST ANNA CATHOLIC CATHEDRAL TOWN OF BAR VINNYTSIA REGION STATE OF UKRAINE BY VIKTOR O LEDENYOV 20150805, Author Viktor O. Ledenyov, [https://commons.wikimedia.org/wiki/File:\(002\)_WIKIPEDIA_ST_ANNA_CATHOLIC_CATHEDRAL_TOWN_OF_BAR_VINNYTSIA_REGION_STATE_OF_UKRANE_BY_VIKTOR_O_LEDENYOV_20150805.ogv](https://commons.wikimedia.org/wiki/File:(002)_WIKIPEDIA_ST_ANNA_CATHOLIC_CATHEDRAL_TOWN_OF_BAR_VINNYTSIA_REGION_STATE_OF_UKRANE_BY_VIKTOR_O_LEDENYOV_20150805.ogv), Date 28 January 2016, Source Own work, This file is licensed under the Creative Commons Attribution-Share Alike 4.0 International license.
- Fence, Poly by Google, <https://poly.google.com/view/atIKFmLzf9->, #enclosure #rail #barrier #barriade, Oct 23, 2017, Objects, Uploaded 3D Model, Public / Remixable (CC-BY)
- Ange, Yves de Bourges, https://poly.google.com/view/6_2EtKz_Ax4, Mar 8, 2018, Uploaded 3D Model, Public / Remixable (CC-BY)Scenery, Pixabay, <https://www.pexels.com/photo/clouds-daylight-forest-grass-371589/>, Source: pixabay.com, CCo License, Free for personal and commercial use, No attribution required
- File:Normal map example - Map.png, Julian Herzog, https://commons.wikimedia.org/wiki/File:Normal_map_example_-_Map.png, This file is licensed under the Creative Commons Attribution 4.0 International license.
- Windmill, Poly by Google, <https://poly.google.com/view/ctIRaTM3zXu>, Oct 23, 2017, Architecture, Uploaded 3D Model, Public / Remixable (CC-BY)
- Barn, Poly by Google, https://poly.google.com/view/0QTh_KUZRYE, Oct 23, 2017, Places & Scenes, Uploaded 3D Model, Public / Remixable (CC-BY)
- Penguin, Poly by Google, https://poly.google.com/view/fBXvsC6pe_V, Sep 29, 2017, Animals & Pets, Uploaded 3D Model, Public / Remixable (CC-BY)

- Grass, Mike Fallarme, <https://www.pexels.com/photo/background-bright-close-up-environment-413195/>, Pexels License, Free for personal and commercial use, No attribution required
- Camaro ZL1 2017, Kris Tong, <https://poly.google.com/view/7bF7UVAoYRG>, A red camaro ZL1 2017, Jul 20, 2017m Transport, Made with Blocks, Public / Remixable (CC-BY)
- Terrain photo, icono.com, <https://www.pexels.com/photo/479849/>, Pexels License, Free for personal and commercial use, No attribution required

13

Bibliography

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN 9780990582908.

Ken Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 287–296, New York, NY, USA, 1985. ACM. ISBN 0-89791-166-0.
DOI: [10.1145/325334.325247](https://doi.org/10.1145/325334.325247).

Ken Perlin. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 681–682, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. DOI: [10.1145/566570.566636](https://doi.org/10.1145/566570.566636).

Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975. ISSN 0001-0782. DOI: [10.1145/360825.360839](https://doi.org/10.1145/360825.360839).

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007. ISBN 0521880688, 9780521880688.

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996. ISBN 0-387-94676-4.