The Development and UI Design of an Interactive Game Map

Tomasz Zawadzki, Korneliusz Warszawski, Slawomir Krezel and Slawomir Nikiel

CONTENTS

12.1 Creating Virtual Cities 200
   12.1.1 Developing UI, City Layers and Simulations in Unity: An Example of the BCD project 201
12.2 Terrain: An Asset for a 3D Game Map 207
   12.2.1 Terrain Representation 207
      12.2.1.1 Height-Field 207
      12.2.1.2 Voxel-Map 208
   12.2.2 Array to Vector Conversion 208
      12.2.2.1 Elevation Model 209
      12.2.2.2 Volumetric Model 209
   12.2.3 Terrain Geometry 209
12.3 Creating Procedural Assets in Unreal Engine 4 210
   12.3.1 Interactive Tree Creator 211
      12.3.1.1 Main Concepts 212
      12.3.1.2 Root, Branch and Sub-Branch Placement 212
      12.3.1.3 Branch Intersection Blending 214
      12.3.1.4 Leaf Placement 215
      12.3.1.5 Fruit Placement 217
      12.3.1.6 Wind Weighting 218
      12.3.1.7 Automatic Billboard Generation 218
12.1 CREATING VIRTUAL CITIES

In 1980, a virtual model of the entire Los Angeles basin was built. The model was used to interactively fly, drive or walk through the city. A virtual scene was constructed by combining aerial photographs with street level imagery and three-dimensional (3D) geometry to create a realistic visual simulation of the dense Los Angeles urban environment, detailed enough for the graffiti on the walls and signs in the windows to be legible (Jepson, 2006).

The Helsinki City Simulator contains a virtual model of the Helsinki city center and a powerful multi-channel display system for real-time simulation on a large screen. The purpose of the simulator project was to build a realistic vision of the future city center as it is planned today. For architects and planners, a virtual model is a platform to test and improve their design. For city residents and politicians, the simulator is an easy and very illustrative way to walk and fly through the future city. It provides a good basis for exchanging opinions on a future design (Suomisto, 2001).
Virtual London was produced using GIS, CAD and a variety of new photorealistic imaging techniques including photogrammetric methods of data capture. The core model is aimed to be distributed via the Internet utilizing techniques to optimize large urban data sets for broadband distribution. There are several vendors who have contributed money and data or donated software to this big project, including CASA, Greater London Authority, Ordnance Survey, Infoterra, ESRI, London Connects (Batty et al., 2000).

The idea of the VR Beirut 3D project was to develop an interactive urban design tool which could be used to consider a building footprint and massing options, as well as maintaining a record of floor space and the proposed land use by parcel, block and sector (Horne, 2004).

12.1.1 Developing UI, City Layers and Simulations in Unity: An Example of the BCD project

The Building City Dashboards (BCD, 2016) project, shortly called BCD, is currently under development in the Department of Geocomputation at Maynooth University in Ireland. A desktop version of the project was built in the Unity game engine and it can be accessed by launching an executable file from the build. There were a few stages in project development including: importing the 3D model (in .FBX format), adding orthophoto map and city layers, developing the User Interface (UI), including buttons, sliders, checkboxes, real-time simulations, navigation modes and other interactive elements. The Dublin 3D city model was imported to the project and placed in the center of the scene (0,0,0). The same was done with orthophoto which let the buildings correspond with the map. City layers were exported in .FBX format from City Engine [2] and imported into Unity [3] in the same place (0,0,0). During export, it was not possible to keep attributes joined with the mesh and we proposed a parser to read .CSV data attributes directly in Unity. Each city layer group node contained all the objects assigned to it. To recognize whether the user clicked on one particular 3D object, it was necessary to assign a script called MapClickDetector which was based on IDs provided in the mesh name (Mesh Filter) and displayed corresponding attribute IDs from the row of the CSV file (Figure 12.1).

To make it work, Mesh Collider had to be added to each game object (single mesh). In addition to that, highlighting would only work when the mouse was over the mesh. It was important to uncheck Convex parameter, otherwise an invisible box was drawn over the shape and despite the
mouse not touching the shape, it still highlighted. This small fix helped to sort it out (Figure 12.2).

The BCD project contained few types of interactive UI menu elements. Buttons were created to facilitate interaction with 3D Dublin city. The most used script called *Enable_Disable* was facilitating showing and hiding game objects in the scene by use On Click () events (Figure 12.3). Based on this approach, it was possible to control all the 3D objects and city layers in the scene.

For planning purposes, it was important that we could hide and show the roads, rivers, bridges, trees, lamp posts, including access to underneath city layers. Objects were grouped into those five groups and controlled by the Development button. Another issue was to add some buildings to the scene which could be activated in runtime. Two free 3D models were loaded into Unity and hidden by default. In planning mode, you can activate them and place in the scene, with their position fixed. Planning mode

---

**FIGURE 12.1** The view from the project and parser window in Unity.

**FIGURE 12.2** Random shape with *Convex* parameter checked (dark gray line outside the shape).
can be utilized by switching between two viewpoints modes added to the scene from the top menu (Figure 12.4).

**BCD WebGL** [4] has two navigation modes: **FPP** (first-person perspective) and **Bird’s-eye View**. **FPP** mode uses **Character Controller**—a default Unity component where you can set up your avatar size, speed and other parameters for locomotion. The second mode—**Bird’s-eye View** explores the city from a higher distance—it uses **Mouse Orbit** script which refers to a center pivot and has 360 degrees rotation with zoom in and zoom out. It is only possible to switch between the two modes (Figure 12.5).

Real-time simulations were added to the project: day and night simulation—correlated with shadow and sun trajectory simulations in a day
cycle. By attaching Directional light to the Slider component (default Unity component), we could control the position of the Sun in the scene. The initial value on the Slider was set up to 0 and Min and Max values ranging from −90 to 90 degrees. RotateWithSlider script with adjustRotation function was necessary to update the Sun position based on the current cursor position on the Slider.

The Flood Resilience (FR) controller works in a very similar way but instead of attaching Directional light to the Slider, we attached the Visualization Controller (VC)—a game object. The VC contains the following: Water plane, Water Height Slider and Water Height UI. From Water Height Slider we refer to the slider in the FR game object. In the Slider, we added the Flooding script with the AdjustHeight function enabled and a Min starting value equal to 0 with a Max value equal to 5 meters. Water Height UI refers to Dynamic Label where we added the Text Mesh Pro UGUI component to be able to display the water level in a text box (Figure 12.6).

On the left side there is access to the Main menu which can be shown and hidden at any time. The menu was created from a set of standard

FIGURE 12.5  Two modes for navigation.

FIGURE 12.6  Left: shadow simulation based on sun day and night cycle. Right: water level simulation.
Development and UI Design of an Interactive Game Map

buttons and image components in Unity. Only this element is animated in the scene. The menu contains eight categories of data which store city data layers (originally. SHP files, but now in .FBX format). Each data category only contains a few sub-categories of data sets and is in early stages of development.

Project logotypes and navigation info were added using Image components. Navigation info changes in real-time based on a user’s interaction and has three main descriptions in the bottom-right of the project. Two of them are related to navigation modes (FPP or Bird’s-eye View) and one to the Viewpoints. In Viewpoint mode, there is no movement and it is to be only used to showcase planning purposes.

Another useful feature was to add a Compass to the project. Only one part of the Compass is animated (Arrow), the rest staying as a background. The Arrow has its own pivot point in the center of the Compass and by default it was set up to the North. To make it work we had to add a Button component to the Arrow image component and an On Click () event assigned to the Main_camera which the Compass refers to. Another step to finally connect it was to select Compass script from the list with the SetNorthDirection function assigned by default.

BCD WebGL contains some icons to enhance the user experience: hide all icons to close all the open windows by a single click and an exit icon—to close the application. On the bottom window, the real-time date and time component was added (Figure 12.7).

Nowadays, game engines like Unity can be used to build games like viewing experiences, even if they do not refer to the game industry, as

![Image of menu and Compass](https://example.com/image.png)

**FIGURE 12.7** Menu and other BCD WebGL UI components.
with the BCD project (Figure 12.8 and Figure 12.9). The characteristic of the project does not change how the UI interface and other interactive elements should be built. However, we are still expected to use game objects, but now for other purposes.

![FIGURE 12.8 BCD WebGL project with an interactive UI FPP (First Person View).](image1)

![FIGURE 12.9 BCD WebGL project – Dublin city layers overview BEV (Bird’s-eye View).](image2)
12.2 TERRAIN: AN ASSET FOR A 3D GAME MAP

The terrain asset used in open world environments is responsible for maintaining an appropriate scale and proportion of objects placed in the scene. A more accurate terrain model increases the visual acceptability and leads to better immersion.

Since the works of Mandelbrot (Mandelbrot, 1982), procedural methods have had a very important function in computer-assisted terrain modeling. Those methods range from basic fractal algorithms to complex techniques producing structures that include erosion influence, rivers, canyons and coastlines (Smelik et al., 2009; Galin et al., 2019).

The manual preparation of a terrain asset is time consuming and in large projects is totally ineffective. A better way is to support it by one or several procedural methods. In this section, we focus on the general preparation of the terrain asset for further use by terrain generation techniques.

12.2.1 Terrain Representation

Terrain models can be categorized into two main types with advantages and weaknesses. The simplest is a model based on elevations, which is referred to as a height-field or height-map. Elevation models can define the geometry of the terrain surface but in general form cannot be used to describe the internal structure of geological materials. Models are also limited to terrains without arches, rock shelves or caves, which cannot be described by height structures. The complex terrain models without these restrictions are based on volumetric representation and are referred to as voxel-map. The voxel models can define both the geometry and the subsurface structures of landscapes. Unfortunately, the methods based on the volumetric model are expensive in memory demands and computational complexity. Hybrid models can also be isolated by trying to accumulate positive aspects of both main representations, while minimizing their weaknesses (Benes and Forsbach, 2001; Smelik et al., 2009; Galin et al., 2019).

12.2.1.1 Height-Field

The common structure to represent the terrain asset in the memory of the computer is to use a height-field (Smelik et al., 2009; Galin et al., 2019). It is defined as a discrete two-dimensional (2D) model over a regular grid and consists of a collection of heights records of the landscape (Figure 12.10). The structure is also used in Geographic Information Systems (GIS) – referred to as the Digital Elevation Model (DEM).
12.2.1.2 Voxel-Map

The complex landscape is represented in computer memory by a voxel-map (Galin et al., 2019). It is defined as a 3D model over a regular grid and consists of a collection of terrain data such as materials, hardness, density and so on. In the simplest form, it takes the form of a Boolean array, where 1 means a solid material and 0 means the air or water (Figure 12.11).

12.2.2 Array to Vector Conversion

In practical implementation, the array representation can be described as a one-dimensional vector in both elevation and volumetric models. In general, the vector notation is faster, and mostly it requires only one loop to process the model in the steps of the terrain generation procedure.
12.2.2.1 Elevation Model
Conversion between dimensions can be performed by the following procedures.

Let us assume that $H$ is a $m$-by-$n$ height-field, where $m$ is the number of columns and $n$ is the number of rows. Let us also assume that $H$ is represented as a vector in column-based notation.

When $k$ is an index of $H$, then indexes of the column ($i$) and row ($j$) can be acquired as follows:

$$i = k \mod m$$
$$j = \text{floor}(k/m)$$

In the opposite direction, the cell data pointed by column and row indexes can be acquired as follows:

$$H[i, j] = H(i + j \times m)$$

12.2.2.2 Volumetric Model
Let us assume that $V$ is a $m$-by-$n$-by-$q$ voxel-map, where $m$ is the number of columns, $n$ is the number of rows and $q$ is the number of pages. Let us also assume that $V$ is represented as a vector in column-based notation.

When $k$ is an index of $V$ then indexes of the column ($i$), row ($j$) and page ($k$) can be acquired as follows:

$$i = a \mod m$$
$$j = \text{floor}(a/m) \mod n$$
$$k = \text{floor}(a/m \times n)$$

In the opposite direction, the cell data pointed by column, row and page indexes can be acquired as follows:

$$V[i, j, k] = V(i + j \times m + k \times m \times n);$$

12.2.3 Terrain Geometry
In game design, the procedural methods for terrain geometry are faster and in general a better way to generate data for terrain assets than simulation-based techniques. In most cases, the reconstruction of real geological formation is not the goal of projects.
There are many forms and variants of landscape modeling procedures, starting with noise or fractal geometry (Smelik et al., 2009; Galin et al., 2019) and ending with genetic algorithms (Saunders, 2006) or machine learning techniques (Guérin et al., 2017).

We would urge readers to experiment with the techniques, adjust parameters and adapt results to the needs of your projects.

Finally in the process, terrain assets obtained by procedural methods (Figure 12.12a) look fine, but should be textured (Figure 12.12b) and adding some lighting effects should increase the realism of the scene. Next, the asset can be extended by valleys (Figure 12.12c). The valley can be simply painted over the terrain surface. Now, the terrain looks better but it can still be improved. As an option, some of the terrain valleys can be filled with water, which makes the scene feel better (Figure 12.12d).

12.3 CREATING PROCEDURAL ASSETS IN UNREAL ENGINE 4

Incorporating procedural asset creation into the development pipeline allows the achievement of large-scale environments and diverse 3D model libraries, without the need for a dedicated team of artists working on each aspect during scene iterations. Procedural assets are also more convenient to modify and iterate on when using parameterized asset properties. After defining procedural generation rules, it is easy to change an asset’s look and swap its elements, which, for example, allows the reuse of the same asset in different projects, even with totally different art styles.
This section will concentrate on describing two procedural asset generation tools which were made in Unreal Engine 4, using the Blueprint visual scripting system.

12.3.1 Interactive Tree Creator

Trees are one of the most common assets that can be encountered in games, especially when it comes to open world titles. Many game projects need a set of different 3D tree models – and sometimes, depending on game mechanics, there is a need for things like tree chopping, smoothly changing wind strength/direction, falling leaves or other dynamic features. It is especially prevalent in survival games or projects where interactive nature objects play an important role in the game loop.

Interactive Tree Creator is a tool that allows the user to create interactive 3D trees, all inside the Unreal Engine 4 (UE4) editor, using procedural generation logic made in the UE4 Blueprint system. It is available on the UE4 Marketplace, along with a video preview for more insight (Figure 12.13).

The user can define many tree properties and immediately see the updated 3D tree preview. Parameters are grouped into separate sections for tree trunks, branches, leaves, sub-trunks, roots and fruits. The default preview scene contains a wind preview object that controls the wind direction and strength – this shows how the generated tree will react to different wind conditions. User can tweak this by changing the Wind Weight Falloff parameter.

![Interactive Tree Creator](image-url)

**FIGURE 12.13** Fragment of tree editor in the Unreal Engine viewport.
12.3.1.1 Main Concepts
The tree generation process is based on using UE4 Spline Components to define the shape and placement of roots, trunk, branches and sub-branches. Spline points are controlled by an extensive set of parameters that are exposed for the end user to tweak.

All the parameters can be randomized based on seed values, which can result in a different tree each time a seed value is changed. It is also possible to control the spline points individually by hand, so full artistic control can be retained despite the procedural nature of the tool.

12.3.1.2 Root, Branch and Sub-Branch Placement
Spline Component allows the user to specify a set of subsequent points in 3D space to create curves with controllable tangents. It also provides GetLocationAtSplinePoint and GetLocationAtTime functions that allow the user to access location coordinates along a specified spline distance – this is used to place branches along the trunk and add sub-branches on parent branches.

By default, roots are placed on the lower section of the trunk spline, but that can also be overridden if there is a need for surrealistic tree types.

Branch orientation is also based on spline data, in this case on trunk spline direction at a specified distance. By combining the obtained location and rotation values, the system spawns another spline that is used for roots, branches or sub-branches (Figure 12.14).

Spline Mesh Component is used to generate 3D mesh geometry along a specified spline. For the trunk and branch generation, a modular uncapped
cylinder mesh is used to distribute it along the spline while respecting all the spline tangents and scale values (Figure 12.15).

Besides the spline-based placement, the user can control the branch number and other placement rules, for example, the yaw rotation method (Figure 12.16).

FIGURE 12.15 Wireframe view of modular trunk mesh with an example along-spline distribution.

FIGURE 12.16 Examples of default yaw rotation methods for branches.
There are also parameters for limiting the individual rotation axis (Roll, Pitch and Yaw values), so it is possible, for example, to generate trees that do not follow the natural growth behavior rules and have branches facing downwards or twisted in a specific way.

It works by getting a main parent branch spline direction, using it to rotate the child-branch rotation vector around it and converting the result to a regular Rotator value type, so it can be used with the final Transform value for a branch (Figure 12.17).

12.3.1.3 Branch Intersection Blending

Each branch has a special start section, where the mesh is scaled at its base to align to the parent branch or trunk more naturally. This solution helps
to improve the look of the trunk – branch transition, but the intersection area is still visible due to the 3D meshes that intersect with each other and have different vertex normals.

To solve this issue, branches are vertex colored at the base. Then, the bark material applies a dithering effect on that region, using the DitherTemporalAA material function. Dithered opacity is better for performance than regular transparency (it also works better under certain specific lighting conditions), since it uses a checkerboard-like pattern with 0–1 values to determine if a pixel should be opaque or not, instead of a more costly alpha blending (Figure 12.18).

This creates an effect where branches visually blend with trunk and parent branch textures at the intersection area. It eliminates the harsh intersection and from a certain distance it looks like a naturally developed transition (Figure 12.19).

The difference is especially prominent when using bark textures with strongly visible patterns. Scaling the UV coordinates can also lead to more or less visible seams and a stronger or lesser transition blending effect.

12.3.1.4 Leaf Placement

Location and rotation values for leaves are calculated using the same spline functions for getting spline point location/direction. Every branch can have a user-specified number of leaves, using a custom leaf layer system that allows for different leaf layers if there is a need for more leaf types growing on the same tree (Figure 12.20).

![DitherTemporalAA blending inside bark material graph.](image)
FIGURE 12.19  Branch intersection blending example.

FIGURE 12.20  Fragment of leaf placement parameters.
By default, the leaves are distributed along the parent branch spline with uniform spacing to achieve a natural leaf coverage. The user can override this by using many parameters for leaf placement.

Any mesh can be used as a leaf. The standard approach is to use simple plane-like meshes with alpha-tested material to reduce the vertex count on the final tree asset. One of the drawbacks is increased shader complexity in scenes with many trees, caused by overdraw (many alpha-tested meshes overlapping each other through their transparent areas). To minimize this, leaf meshes are trimmed in a way that they fit leaf texture as closely as possible, without increasing the vertex count too much. The goal is to reduce the transparent area (coming from the assigned leaf texture alpha channel or opacity mask) on the mesh (Figure 12.21).

There are more leaf growth behavior settings in the ITC tool. For example, leaves can grow smaller at the top of trees or be more up-facing the higher they grow – this is useful for creating certain tree types, for example, some conifer trees.

12.3.1.5 Fruit Placement

Procedural placement methods can also be utilized to add fruits on generated trees in the correct places. The system calculates fruit locations by getting mesh data from leaf planes and offsetting mesh surface points by fruit mesh pivot points. The final effect is approximated (especially when looking at leaves during GPU wind simulation in the vertex offset shader), but from a certain distance it looks convincing enough for use in most cases (Figure 12.22).
12.3.1.6 Wind Weighting
Wind is simulated using a GPU vertex offset inside the leaf and branch shader. To make the trees behave realistically, vertices need to be weighted for different wind intensity values, depending on height along the tree and progress along the parent branch. This way, tree branches can sway differently based on their location when affected by the wind.

The system writes this data into vertex colors of branches and leaf meshes, using 0–1 value, where 0 means no wind effect and 1 corresponds to full strength wind effect. Then, the material can read it via the *Vertex Color* node and apply the correct vertex offset for current wind strength and direction, using the *Lerp* node (linear interpolation) (Figure 12.23).

12.3.1.7 Automatic Billboard Generation
Billboard is a term used to describe a simple version of an asset that can be used to mimic mesh geometry when viewed from a great distance – usually it is a flat plane mesh with assigned texture representing the original asset
model. It is important to reduce vertex count in more complex scenes (e.g. dense forest scenarios) and using billboards as distant mesh versions at largest LOD (level of detail) helps with that.

The Interactive Tree Creator allows the generation of billboards automatically for each created tree. The system measures tree dimensions in all three axes ($XYZ$), comparing each result and based on maximum bound extent it applies proper dimensions for the billboard mesh plane. This way the tree billboard plane matches the true tree geometry.

The tree image is captured via the **Scene Capture 2D** component that writes the result into a **Render Target** texture. The capture component needs to be located at an accurately calculated position, in a way that it encompasses the whole tree in its camera view, using orthographic projection. The component is also tweaked to capture only the specified objects, in this case only the tree that needs to be captured. The rest of the scene is discarded in the scene capture rendering (Figure 12.24).

Scene lighting is altered before capturing the subject to eliminate as much lighting information as possible. Evenly lit, neutral billboard texture is required to make it usable under different lighting conditions.

There is also a possible improvement for that which requires generating dedicated normal maps for each billboard. Having normal maps on billboards can help to achieve realistically lit distant trees with an additional sense of depth.
12.3.1.8 Preparing the Trees for Use

Procedurally created trees can be converted to regular static meshes, which allows them to be used with UE4 instanced mesh foliage tools. This is necessary to achieve good runtime performance and to be able to place trees in 3D scenes in a convenient way, either by hand (by using the Unreal Engine 4 Foliage tool) or procedurally.

Using the Merge Actors tool, procedurally generated trees can be saved as static meshes while retaining all the necessary data like vertex colors.

12.3.1.9 Force Reaction and Dynamic Wind

Trees generated with the Interactive Tree Creator contain additional information (stored, for example, in vertex colors) that is used by the main tree shader to apply different forces and make the trees react accordingly (Figure 12.25).

The default system allows for control of the dynamic wind speed and direction, point impulse forces and constant radial forces. Besides dynamic, smoothly changing weather conditions, these features also allow for things like bomb shockwaves or helicopters flying over trees while affecting the branches and leaves in a convincing way.
The system knows where to apply force points through the global material parameter collection. Location, radius and strength of a force is passed from tree Blueprint logic to the global parameter collection, then it is accessed by the main tree material shader (Figure 12.26).

**FIGURE 12.25** Point force example, affecting trees within a specified radius.

**FIGURE 12.26** Force reaction logic flow.
Blueprints also control the duration of forces, and when it is time to turn one off, they gradually decrease the force strength value via Timeline with custom curve (Figure 12.27).

12.3.1.10 Tree Chopping, Falling Fruits, Leaves and Debris

Each procedurally created tree is assigned to a custom Foliage Component, where all the tree specific data is stored. This allows the implementation of additional game mechanics like tree chopping or wood and fruit gathering.

The tree chopping system supports many chopping progress stages, where each tree type can have a different HP (Health Points) value. Based on that, the developer can set these values in a way that each chopping hit will gradually carve in the tree surface, resembling a real chopping process. To achieve that, tree Blueprint generation logic saves a separate set of data: top tree part, bottom tree part and parameters for trunk area where chopping should occur. Then the Foliage Component swaps the tree mesh to chopped version and adjusts the chopping area to match current tree HP value (Figure 12.28).

If the HP value reaches 0, the tree is marked as chopped and it starts simulating tree fall, using the Unreal Engine 4 physics engine. Falling trees are controlled by dedicated Blueprints where the system waits until it hits any surface by checking current tree velocity and comparing it with the previously recorded value. If it is drastically different, then it can...
assume that the tree landed, and it is the right time to play impact animation (Figure 12.29).

Upon impact, the tree can spawn other physics objects defined by the user, for example, bark parts, little twigs, leaves or fruits. There is also a slot for a particle system that can be used to add dust particles to enhance the tree impact effect.

![Tree chopping, falling leaves & debris](image1)

**FIGURE 12.28** Tree chopping in game.

**FIGURE 12.29** Logic for checking if the falling tree hits the ground.
Besides the fall impact, trees can also drop leaves while being in an idle state or while a player chops the trunk. These effects (for idle leaf falling) are calculated only when a player is near a tree for performance reasons. The simulation cull distance is controllable.

Another way to make leaves, fruits or debris fall from a tree is to use the included force effect Blueprints. When impulse force is deployed near a tree and it is strong enough, it will make it drop a specified amount of leaves, debris and fruit (if the tree has any fruits).

12.3.2 Smart Spline Generator

Some game assets are meant to be used only in specific level locations, conforming to unique scene geometry. For example, vines or other plants that hang from point A to point B. Assets like these can be cumbersome to create, especially when there is a need to modify the underlying level geometry or add a new element that interferes with current asset design.

Smart Spline Generator is a tool that allows the user to procedurally generate assets that automatically align to level geometry. The user can use any custom mesh, so the generator can produce vines, fences, ropes, cables, walls, drainpipes and more. It is available on the UE4 Marketplace, along with a video preview for additional insights (Figure 12.30).

12.3.2.1 Main Concepts

Procedural surface-aligning asset generation works with any collision enabled surface. The user can drag an asset preset from the Unreal Engine 4 editor content browser and drop it into a 3D scene to immediately see the effect. The generator detects surface in a specified direction, over a specified distance, using many parameters exposed from the Blueprint system.

![FIGURE 12.30 Example assets created with Smart Spline Generator.](image-url)
12.3.2.2 Recursive Surface Align Algorithm

A custom algorithm was created in order to implement the surface align feature. After setting the main direction vector and finding the start point using Line Trace functions, it executes a recursive function for surface detection in a previously calculated direction (Figure 12.31).

SetMainDirectionVector allows the user to specify in what direction the asset should generate. The main direction vector is saved as a variable inside the main generator class, so the other functions inside it can access it later. The variable is also used in other parts of the tool, so the value is not passed through the function outputs in this case.

FindStartPoint function performs a Line Trace By Channel, where it detects a starting point from near the surface in a specified direction. It also returns the normal vector of a detected surface, so the next function can use it to align the detection points properly.

DoCheckSegment: most of the surface detection logic is contained in this function. It is recursive, in this case it executes itself if a specified condition is not met yet, for example, if the user wants to generate X amount of spline segments along the surface, it will keep generating until the desired number of segments are created.

After each execution, the function passes new calculated values for inputs StartLocation and RelativeNormal when calling itself from inside. This way, it can use the last calculated point as a new starting point to detect and generate points one after another, progressing through the detected surface.

FIGURE 12.31 Initial logic for surface detection.
Surface detection relies on *LineTrace* functions from calculated locations to check the hit location and normal values to determine further generation steps. In general, the algorithm works as shown in Figure 12.32.

Additionally, the algorithm checks if the specified point amount is already reached. If that is true, it does not execute the function, breaking the recursive execution chain.

The set of points calculated in the *DoCheckSegment* recursive function are passed to the next function which creates *Spline Mesh Component* along the points.

### 12.3.2.3 Vert Count Optimization with Spline Thicken Function

Using modular cylinder meshes as *Spline Mesh Component* sections will provide good results for most cases, but for larger assets, it introduces a significant vertex count increase. To reduce the vertex amount, the system uses the *Spline Thicken* function inside the main material. This function is a part of the Unreal Engine 4 toolset and it allows the use of simple polygon strips instead of cylinders while retaining the cylindrical look (Figure 12.33).

![Surface detection algorithm flowchart.](image-url)
Thickness is controlled inside the material by getting vertex color values from polygon strip mesh. The mesh is painted with the \textit{PaintVerticesLerpAlongAxis} function with 0–1 value that represents thickness. It is saved into the \textit{Red} channel of vertex color. This way the user can control the thickness along surface aligned points, usually it is used to taper vine stems at the end or set the overall volume of stems, cables, ropes and so on (Figure 12.34).

Controlling the stem look via vertex colors also allows to add more visual tweaks inside the material. There is also room for more features that can be controlled using other vertex color channels (green and blue).

\subsection{Merging to Static Mesh}
Assets generated with Smart Spline Generator are ready to use immediately, but it is recommended to merge them into a regular static mesh as a final step. This significantly reduces the draw call amount for each asset by combining all spline mesh sections and, for example, many leaves mesh into one object. It can be achieved with the Merge Actors tool that is available in Unreal Engine 4, under the Developer Tools menu.

Merging an asset into one static mesh also provides a way to easily save generated assets and export them in FBX or OBJ format to external 3D applications for further editing.
12.4  GAME LEVEL DESIGN

Game level is the arena for both players’ avatars and non-playable characters (NPCs). It gives the game player a sense of “Presence” to immerse her/him in the gameplay. Game design includes the planning and integration of various 3D assets that form the entire game “world.” Terrain, buildings, foliage, lights and other elements including HUDs (Head Up Displays) should be thoughtfully composed to ensure maximum playability and fun for the player. This section provides hints on successful game level design.

12.4.1  Game Level Construction: “Presence” Approach

The synthetic environment of the first-person perspective (FPP) game level is created to give the impression of “being” in the multisensory three-dimensional space. The phenomenon of immersion is closely related to the quality of experiences experienced by means of sight, hearing and touch. In recent years, there has been a rapid development of computer graphics techniques enabling hyper-realistic depiction of the visual space. The introduction of emotional visuals and narratives increases the sense of “Presence” or immersion of the game player. The aims of this section are:

- to understand the potential to use ideas from “Presence” space and place when considering immersion of the game player;
• to understand different properties of visual space which can impact upon game level design; and

• to learn about a specific method of evaluation.

12.4.1.1 Presence and the Concepts of Space

For the first time, the term “Presence” was mentioned by Minski (Minsky, 1980) as an experience of being in a remote environment. In literature, “Presence” is most often given as “behaviour in a virtual environment as in the real world” and refers not only to spatial relations but also to non-physical elements of perception and interaction (Riva, 2003). The definition of “Presence” as a phenomenon gives it as “an effective replacement or enrichment of reality perception by virtual stimulation” where efficacy is confirmed by user behavior analysis in physiological, psychological and behavioral aspects and comparison of results obtained without stimulation (EU FP6 PRESENCCIA). Effectiveness depends on the quality of interaction and information transfer along with its context. Another definition of “Presence” as a research area is focused on understanding and controlling the “experience” (in the perceptual and emotional sense) of being “somewhere.” “Somewhere” is understood as a Place and Space and time or as another person. “Experience” is provided through technologies such as VR (virtual reality), AR (augmented reality), and Telepresence. Other definitions of “Presence” (abridged version of the remote presence of Telepresence) include a psychological state or a subjective perception in which, despite synthetic stimulation, the user is unable to partially or completely recognize the use of technology in contact with the environment. The environment is observable, stochastic, sequential, dynamic and is inherently a continuous process (ISPR: International Society for Presence Research: http://ispr.info/). Space is an ontology for three dimensions, taxonomy and linguistics that organize it. The American researcher Edward T. Hall dealt with determining how we behave in various socially and culturally defined spaces-distances (Hall and Pellow, 1996).

Proxemics define the invisible personal space of a man whose reach is shaped by a culture specific to a given person. It determines not only distances, but also the boundaries that divide us from others (intimate, personal, social, public). For example, Jandt (see Table 12.1) specifies the dimensions and method of interpersonal communication in these spaces (Jandt, 2007).
Several researchers tried to define the relationship between the concepts of space and the feeling of presence/immersion. According to Slater and Wilbur we can be immersed within the movie, game or virtual reality to the extent that displays can deliver an extensive, surrounding and vivid illusion of reality to our senses (Slater, 1997). The more we forget about the medium delivering stimuli, the more immersed in this environment we are. Then, we experience a high feeling of presence when we have a more vivid memory “as if we were there.” Based on the results of research related to IP City and Benogo projects (McCall, 2008), it is possible to determine how places are perceived in both real and fully synthetic environments. The “place” is characterized by several properties, the most prominent is the association of space and meaning. The place is interesting when it has a certain function. The places can be connected to form environments. We are particularly interested in highly responsive environments, since they are the backbone of good game level design. Such environments are characterized by permeability (usability), variety (of associated actions), robustness (multiple actions and uses), visual appropriateness (we can easily recognize type of the place), aesthetics and visual richness (with consistent style) and legibility (we can recognize all assets).

12.4.2 Level Design: Game Genres and Levels
The definition of “game level” varies greatly according to the game type. It usually refers to the game world of arcade side-scrollers, puzzles, adventures, flight and car simulators, FPS (first-person shooters) and RPGs (role-playing games). These games have distinct playing areas which are referred to as “game levels.” These areas can be constrained by geography (maps), by geometry (arenas) or by amount of gameplay that must be completed before the player is granted access to the “next” (usually more
“demanding”) levels. Some games (retro/arcade) take place entirely on one level, other have some variations of the game area, and they can be linked into maps, mazes or tracks (for racing games). There are also so-called “building” games where the level is designed and built by the player (Minecraft and Fortnite are the most popular examples). The base levels for such games are usually randomly generated and it is the player’s responsibility to construct the level during her/his playing. This chapter deals primarily with games that use pre-built levels that have a great impact on user experience and gameplay.

Different types of games use different visuals. Table 12.2 illustrates different viewing approaches (and level construction paradigms) according to game imaging technology. The chapter contents are mostly devoted to 3D games, although some ideas can be relatively easy ported to 2.5 and 2D games (Rogers, 2014; Rouse, 2001).

### Table 12.2 Visuals, Types of Games and Camera Narrations

<table>
<thead>
<tr>
<th>Game engine</th>
<th>Typical Games</th>
<th>Description</th>
<th>Camera</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>Arcade/retro</td>
<td>Flat images, that can be arranged in different layers. The main visual tools are sprites and image textures.</td>
<td>Flat, orthogonal. No perspective applied.</td>
</tr>
<tr>
<td>2.5D</td>
<td>Ecosystem simulations, RPGs, strategies, combat arcade</td>
<td>“3D”-like looking images put on a plain background. The driving technology is still sprites/impostors and textures.</td>
<td>Isometric camera with “God’s eye” perspective, other perspectives can be applied (cavalier, etc.)</td>
</tr>
<tr>
<td>3D</td>
<td>First-person shooters, fighting, sports, simulations, RPGs</td>
<td>Fully 3D assets, the game is displayed on a flat screen. The underlying game engine operates on a plethora of graphic object, including 3D meshes, 3D sprites, particle systems, textures and lights.</td>
<td>Moving/floating camera, first-person perspective, third-person perspective. Cinematic quality requires “camera operation” as it is done in classical movies.</td>
</tr>
<tr>
<td>3D XR</td>
<td>RPG, first-person shooters, puzzle/escape rooms, music/rhythm games</td>
<td>Fully 3D assets, the game is displayed in on head-mounted displays or VR goggles. The underlying game engine operates on a plethora of graphic objects, including 3D meshes, 3D sprites, particle systems, textures and lights.</td>
<td>Usually first-person perspective only. Stereoscopic imaging is necessary to realistically depict the game “universe.”</td>
</tr>
</tbody>
</table>
Different types of games require different approaches to game level design. The underlying visual technology (performed by the game engine and reflecting the gamer’s point of view) determines the level flow. There are basically two groups of games: linear (action/adventure, puzzle solving, role-playing and racing) and non-linear (strategy/simulation, sports and shooters/death-match). Whether it is a linear or non-linear game, various game level components will be arranged in a different manner (Costikyan, 2015).

12.4.3 Components of a Level

Once the idea of the story describing the game and the type of game have been decided in, the next task is to actually design and create the game levels. The goal of each level is to provide a highly engaging experience and maximum fun for the player. In this pre-production stage, we need to remember that each level should be played differently from others. Before construction of the level, the team/designer should break down the various gameplay components of the game. One of the approaches is to break down the components of a level into elements associated with geometry, actions, asset and sounds (Rouse, 2001).

12.4.3.1 Geometry of the Level

Physical space consists of various elements that give meaning (functionality) to the place (Figure 12.35). When applied to the game level, appropriate use can enhance playability of the game and clearly state the objectives of game design.

**FIGURE 12.35**  Different geometry elements of physical space.
All elements can be composed to a game level map (see next section) and illustrate game narratives visually. Paths declare, then, routes for possible movement of a player’s avatar and NPC, Districts offer exploration areas and can also serve as a duel arena, Edges funnel the player to the predetermined goal, Landmarks offer memorable visual clues and help with player’s orientation and Nodes are goals and assets to be completed/collected during gameplay.

12.4.3.2 Game Level Description
Before placing all assets on the game level scene, it should be thoughtfully designed. Time spent on design pays off during final composition and saves effort in the further game testing phase.

12.4.3.2.1 Description of a Place
Concept of the game level design can be outlined by several means, including text, storyboards, concept arts and mood boards. Game documentation agreed upon at the previous game development stages usually contains names of Section/Level/Scene and its textual description including physical and audio appearance. A sketch of the background, or mood board are very useful (Figure 12.36). Foreground objects and characters (see Section 12.4.3.2.3) may be depicted too (Schwartz, 2005).

12.4.3.2.2 Level map
Level maps help to clarify the definition of the landscape and playable areas/districts. They include positions of all-important assets and (simplified/iconized) physical elements. A simple legend

FIGURE 12.36 Sample moodboard for the post-apocalyptic game. (Created by Anna Wieszczeczynska as a part of student’s project at the University of Zielona Góra, Poland, 2014. All permissions granted.)
attached to the map can help with understanding the construction of the level (Figure 12.37). Additional descriptions of action paths and asset roles (Section 12.4.3.2.3) is welcome too.

12.4.3.2.3 Assets  Digital assets are all elements of the game. Most of them are defined by geometry, however, they include digital sounds and AI algorithms. Speaking of geometry, digital assets are graphic elements that can be 2D or 3D meshes, with or without rigs (internal skeleton). Graphic elements also include textures/materials that define their appearance, lights/shading (responsible for spatial properties of the scene) and various special effects (fire, smoke, explosions, water etc. where we cannot strictly determine underlying geometry). We can generally define different types of assets specifying their role in the game and position (node) or path on the game level map. While placing active elements on the map we can use the distances defined in Table 12.1 to evoke desired emotions (Freeman, 2003; Koster, 2013). A sample description of assets is provided in Tables 12.3 and 12.4.

12.4.3.2.4 Sounds  Music and sounds add mood and sense of physical reality to the game level. While music and generally background sounds are omnidirectional and add some “spice” to the scene, action sounds are tightly connected to physical assets and their performance. When you shoot a shotgun, you should hear the explosion and the sound of hitting pellets. When you walk on the broken glass, you should hear cracks and so on. Such sounds are directional, have location and limited range.
12.4.3.2.5 Technical Design Specification When the early prototype of the game level is about to be ready, it is worth documenting technical details. This saves time and money, especially in cases where the game project is shared with other developers or gamedev team members allowing many people to work at once. If we document further details of assets (type and size of meshes, rigs and textures, definition of data structures and interfaces, parameters of procedual design) and pseudocode describing actions and interactions, it forms a solid background for game development.

<table>
<thead>
<tr>
<th>Color category</th>
<th>Cost</th>
<th>Player/AI control abilities</th>
<th>Combat</th>
<th>Speed</th>
<th>Lifespan</th>
<th>Hitpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>Low</td>
<td>Scout and errand. Can use few weapons.</td>
<td>Low</td>
<td>Fast</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Gray</td>
<td>Medium</td>
<td>Medium combat skills. Can use some weapons. Can transfer health to humans.</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Yellow</td>
<td>High</td>
<td>Good combat ability. Can use all weapons. Strong defense.</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Black</td>
<td>Very high</td>
<td>Strong combat, detect and disable traps and ambushes. Seek and destroy</td>
<td>Very high</td>
<td>Slow</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>

### TABLE 12.3 Action-Type Definitions of Active Assets

<table>
<thead>
<tr>
<th>Color category</th>
<th>Cost</th>
<th>Player/AI control abilities</th>
<th>Combat</th>
<th>Speed</th>
<th>Lifespan</th>
<th>Hitpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>Low</td>
<td>Scout and errand. Can use few weapons.</td>
<td>Low</td>
<td>Fast</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Gray</td>
<td>Medium</td>
<td>Medium combat skills. Can use some weapons. Can transfer health to humans.</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Yellow</td>
<td>High</td>
<td>Good combat ability. Can use all weapons. Strong defense.</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Black</td>
<td>Very high</td>
<td>Strong combat, detect and disable traps and ambushes. Seek and destroy</td>
<td>Very high</td>
<td>Slow</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>

### TABLE 12.4 Sample Asset Classification Based on the Narratives

<table>
<thead>
<tr>
<th>Asset category</th>
<th>Action</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monster</td>
<td>Eat</td>
<td>Jungle</td>
<td>Teeth</td>
</tr>
<tr>
<td></td>
<td>Growl</td>
<td>Trees</td>
<td>Horns</td>
</tr>
<tr>
<td></td>
<td>Hide</td>
<td>River</td>
<td>Scales</td>
</tr>
<tr>
<td></td>
<td>Run</td>
<td>Meadows</td>
<td>Claws</td>
</tr>
<tr>
<td></td>
<td>Sleep</td>
<td></td>
<td>Tail</td>
</tr>
<tr>
<td></td>
<td>Hunt</td>
<td></td>
<td>Eyes</td>
</tr>
<tr>
<td></td>
<td>Scratch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scare</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fairy</td>
<td>Sing</td>
<td>Trees</td>
<td>Wings</td>
</tr>
<tr>
<td></td>
<td>Hide</td>
<td>Bush</td>
<td>Fur</td>
</tr>
<tr>
<td></td>
<td>Sleep</td>
<td>Meadows</td>
<td>Eyes</td>
</tr>
<tr>
<td></td>
<td>Help</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
12.4.4 Aesthetics

Visual emphasis greatly influences “Presence” of the player and her/his gameplay. The way the game is emotionally perceived is strongly related to a level’s appearance. Even if the game is visually gorgeous, we must remember that all elements should support the player in game, help him/her to navigate, engage AI powered NPCs and comply to hardware specification in order to make the game engine run efficiently. To do so, the level designer should balance the appearance of the level with its functionality. The ideal compromise is when the level looks great, actions are smooth and all renders quickly and fit in the narratives of the game. The level designers “collect” their experiences with a given game engine or level editor and make a “workbench” of “tricks” to be applied. Some solutions are not very realistic but they are good enough to be plausible. If the player cannot tell what elements of the game level are “fake” it is okay. This is the basic principle of VFX (visual special effects), to create something that looks like something when it is actually different. The level designer job is similar to a VFX artist! The visuals of the game level have a big impact on the game perception and immersion of the player, it holds especially for the XR (eXtended reality) games including virtual reality and mixed reality games. Plenty of time can be spent on the aesthetics of a game level. The time spent in the creative process is directly proportional to the complexity of the underlying game engine and the given level editor. Some elements like lighting can be endlessly adjusted (the lighting of the scene is beyond the scope of this chapter, we suggest following the rules of digital photography guide books, for example (Donati, 2009). We can, however, use several “tools” and rules to make the game level design process easier, including: motion, location, size, shape, padding (HUD), rule of thirds, leading lines, rhythm-actual, implied, psychical, contrast, hue, saturation and humans/face/eyes. See following examples. Motion and movement are great methods for getting players’ attention guiding their actions. Dynamic elements of the level highlight points of interest and can also serve as “landmarks” (Figure 12.38).

Location and size. Sometimes it is useful to funnel the player through narrow areas when you want to orient her/him towards the goal. By funneling the player’s avatar, you know where the player will be looking. It is particularly useful in VR games and for guiding the player after navigating through open areas (Figure 12.39).
Shape. Characteristic shape can serve as a landmark or grab attention of the game player. The shape should create visual contrast between it and the rest of the environment (Figure 12.40).

Padding (HUD). Head up displays provide current information (status, time, compass, life, ammo etc.) to the player. HUD should be legible in any
level/lighting conditions and should not obscure the game arena during the action (Figure 12.41).

Leading lines. Some visual patterns suggest directions and movement to the game player. The leading lines are the most straightforward approach (Figure 12.42).
Rhythm. Repeated elements of the scene can help to guide the player through the level. (Figure 12.43) Contrast and lighting also help with depicting possible targets to the game player. (The floor in Figure 12.43 is clearly lighter than the surroundings.)

Hue and Saturation. Careful use of hue and saturation is necessary in the design of open spaces (Figure 12.44). So-called atmospheric effects (clouds, fog, smoke) highly influence the mood of the game.
Humans/face. The appearance of humans, in particular, their faces, has a very strong impact on the mood of the game. While humanoid characters are the most demanding assets (requiring rigging, skinning and nowadays motion and performance capture) they certainly pay off during the game. First of all they add a “human” touch to the scene and define proportions of assets, evoke different emotions and moreover they can serve as a guide in helping the game player to complete all the tasks assigned in the game (Figure 12.45).

12.4.5 Elements of Good Level Design
A good game level must balance several things. The game level designer is not only a VFX expert but should balance aesthetics, action, storytelling and technical requirements (texture sizes, meshes, simulations) to make everything work smoothly. All elements are interdependent, and the decisions made are always a kind of trade-offs. There are common measures that can form a “checklist” to test completeness of the game level. Every designer will have her/his own list of “dos” and “don’ts” that is kept in mind. A collection of typical “design rules” which can be applied virtually to any project is presented here.

Player cannot get stuck. The player’s avatar should never become stuck when playing the level. The routes should be designed with no holes to be fallen into without the possibility of climbing out, no objects can block...
the way both for the avatar (always) and NPCs (unless they can be placed and removed by the player) and no doors that fail to open under certain conditions. This requirement should be obvious, but to make it happen, it needs a lot of time to design and test paths and actions on the given level. The best way to achieve game flow in such situations is either to end the game instantly in case of a wrong sequence of actions (if the key is needed to open the door and the player destroyed the key with dynamite...) or to provide alternative solutions/actions. The level must end despite any actions of the game player.

Sub- and side-tasks. As the player plays the game level, she/he should clearly understand how to accomplish some larger goal. To do so, the level should consist of various clearly defined sub-tasks to be completed to archive the main goal. This can be done through various sub-quests, funneling the player through some level areas and hints/positive feedback after the completion of certain tasks.

Milestones and Landmarks. The more levels are “completed” by the player, the harder it becomes to complete them. The player should not be confused while navigating even more complicated levels. The best way to help him/her is to use memorable visual landmarks, characteristic to the level. A good landmark is any unique visual object in the level that the player recognizes next time she/he sees it. As far as exploration is
considered, when the player returns to (respawns at) the landmark, she/he will know it is the location previously visited.

Critical paths. In cases of non-linear level flow, the game needs critical paths to give the player the sense of direction she/he can go in order to complete the level. This can be a physical direction, signs or some more ambiguous goals. The player should have a clear idea of the possible routes in the level. We can use various visual clues as illustrated in the previous sections to help the player move around the level and find the tasks.

Limited backtracking is necessary in games where the game covers a geographically large area. If the player has already explored some extent of the game, it would be a bad idea to force him/her to go through large sections of the level to continue the game.

Multiple choices. Good levels give the game player choices of how to accomplish tasks. Choices do not always mean multiple paths through a level; we can use different assets or alternative game logic to extend available options. We can provide alternative ways to complete the level taking into account, for instance, the easiest and toughest situations that can be faced on the level.

Estimated time to complete the level. The best way to control the level flow is to control the time to complete it. Varying time can induce different “hardness” levels and may force the player to perform faster actions or the opposite, get some rest and explore the district both for “fun” (so-called “Easter eggs”) or to collect necessary assets useful at further levels.

ACKNOWLEDGEMENTS

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 15/IA/3090.

REFERENCES

Development and UI Design of an Interactive Game Map


