MAT JOURNALS
Journal of Android and IOS Applications and Testing
Volume 2 Issue 3

3D Game: Innovating India

Mandar Deulkar, Vedika Basarkar, Mrudula Bangale
Computer Science and Engineering Department, Dr Babasaheb Ambedkar Marathwada University
Aurangabad-431001, India
deulkarm@gmail.com , vedika.basarkar30@gmail.com ,mrudula5.bangale@gmail.com

Abstract
This document gives information about 3D game using Unity game engine and Autodesk maya. This game is totally strategy game. Build your village, gathering of resources and utilize them, earn some money and spend that money on to the development of the village to convert it in to smart city.

Keyword: *3D Game, Assets, Scenes*

INTRODUCTION

The basic development of any country starts from its basic and small structural development; we call it as initial stage as village and the highest improved stage called as Smart city. If your villages are enough stronger to hold basic needs for society then it will automatically possess potential to develop itself up to smart city. India have emerging market, growing economy, enormous projects but still our villages are filled with poverty, lack of resource availability, lack of employment etc. which results in migration of people towards city. This is a Strategic game. We will provide a piece of land to player i.e., a village. Player has to develop that piece of land into a smart city using resources and man power. Player has to build village, train population, gathering of resources and utilize them, earn money and spend it on the development of village. Player will face real time problems.

3D ASSETS IN MAYA

Add a collection of nodes to a specialized Maya asset node for ease of scene management. Assets are particularly useful for streamlining and maintaining workflows while simultaneously allowing the scene author to securely control what aspects of the scene particular artists are allowed to modify.

Some of the things you can do with assets are

- Hide internal non-published s nodes by making the asset a black.
- Pre-plan the organization and attributes of a scene.
- Customize the attributes displayed in the Channel Box and Attribute Editor.
- Lock attributes from being.
- Organize nodes together by function without any effect on performance.
- Substitute parts of your model with other parts while maintaining behavior and animation.
- Reference assets from external files and display them in your scene as proxies.

Nodes contained in an asset are called *encapsulated nodes*.
With assets you can create templates to plan and organize the capabilities and attributes of various parts of the scene before you actually build them. You can also set up assets and publish attributes as you create your geometry and then save a template based on that for future scenes. Either of these methods can be used to set the expectations for future artists working on the asset.

As an example of process management, a rigger working on a character can create an asset with an *interface* of attributes needed for animation. When the model is sent

down the pipeline, the animator only sees the attributes that are key able and exposed for the animation department. This simplified subset of attributes prevents the animator from having to search through (and possibly modify) long lists of non-animation attributes.

Assets are also useful when parts of the model need to be swapped in and out. For example, multiple versions of arms and legs for a robot can be stored in their own files and referenced into a scene using assets without breaking existing hierarchies and animation.

PARTS OF ASSETS

While asset nodes themselves do not appear in the scene view, they do appear in many editors. Additionally, if you select an object encapsulated by an asset in the scene view, the asset is displayed in the Attribute Editor / Channel Box along with their published attributes.

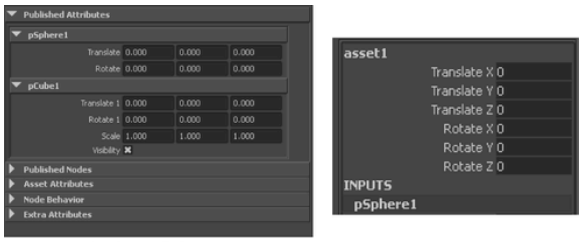
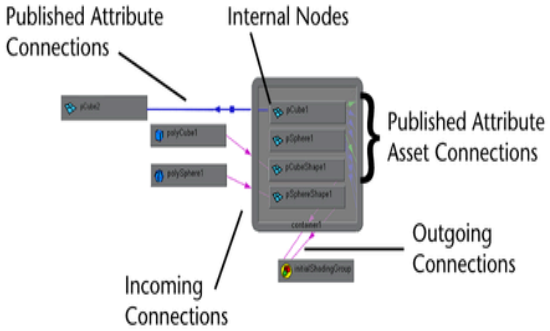


Fig1. *Atribute Editor And Channel Box*

Assets appear in the Hypergraph Connections editor as nodes with beveled edges. If you expand one of these nodes you can see a number of parts:



WORKING WITH ASSETS

When you create an asset, Maya arranges the nodes in a number of ways:

- In the Hypergraph Connections editor, all the internal nodes encapsulated by that asset appear as a single node with rounded borders.
- In the Outliner, all the internal nodes encapsulated by the asset appear under a new asset node.
- A corresponding entry appears in the Channel Box above the Inputs and Outputs.

Types of Assets

In Maya there are two types of assets, and you can interact with each one in different ways.

Asset with transform are assets that have transform properties and can be manipulated in the scene like a group node. You can also parent assets in the DAG hierarchy like any other node. Any node parented to an asset with a transform is automatically placed inside it.

Assets with transform are simpler and allow direct manipulation and thus are most appropriate for nodes that need to be placed in the scene or the DAG hierarchy (for example, geometry or groups).

Advanced assets do not have an associated transform. These assets are not part of the DAG

Hierarchy and the assets themselves cannot be parented in the hierarchy. However, internal nodes can be published as anchors and placed in the DAG hierarchy if necessary.

Advanced assets are most appropriate for collections of nodes that do not need to interact much with the DAG hierarchy, or for collections of nodes that are not in the same DAG hierarchy.

In both cases, you add and modify attributes and make attribute connections to asset nodes (and the nodes placed within

them) just like you would other nodes in Maya.

EXPORT SCENE TO UNITY

To export the scene to your Unity project

1. Select File > Send to Unity > Set Unity Project, then browse to select a valid, local Unity project (the project directory) and click Select.
   (You need to set the project only once. It's not necessary to set the project again for subsequent export operations.)
2. Select File > Send to Unity, then select All (to export the whole scene) or Selection (to export only selected objects).
3. In the Export Selection window that appears, enter a name to save the selection as a FBX file in your Unity project's Assets folder (default) and then click Export Selection.

Maya exports the selected objects or the entire scene to your Unity project. If you have the project open in Unity, you can access the FBX file immediately. To see it in your Unity scene, drag and drop the file from the Assets folder in the Project Browser into the Scene view.

IMPORT SCENE TO UNITY

Unity natively imports Maya Files. To get started, simply place your .mb or .ma file in your projects file folder. When you switch back to unity your objects is imported in unity and will show up in our main project View.

Unity currently imports from Maya:

- We imported with all nodes, positions, and rotations of scale.
- We imported all meshes with vertex colors, normal with and upto two UV sets.
- We can also imported materials with all textures and diffuse colors and multiple colors with mesh.
- Animations with FK & IK.
- Bone-Based Animation.
- Blend shapes.

CREATING GAMEPLAY

There are a handful of basic workflow concepts needed to learn in unity. Once you understood, you will find yourself making games in no time. This section explain the core concepts you need to know for creating unique and amazing gameplay .The majority of concepts required you to write scripts.

1. SCENES: Scenes contain the objects of your game. They can be used to create main menu, individual levels and anything else. Think of each unique scene file as unique level. In each scene, you may place your environments, obstacles and decorations, essentially designed and build scenes.

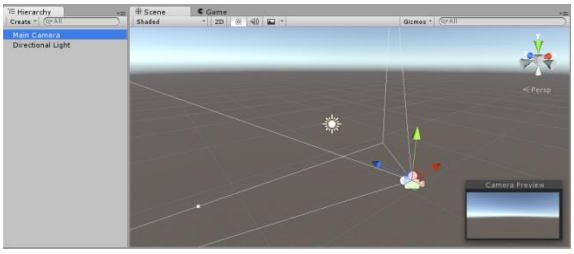


Fig2. *Basic View Of Scene In Unity*

When we created our 1st Scene that time scene was empty expect for default objects- either and orthographic camera, or perspective camera and directional light, depending on whether we started the project in 2D or 3D.

GAME OBJECTS

Every object in your game is game object. This means that everything you can think of to be in your game has to be a game object. However, a Game object can't do anything on its own; you have to give it properties before it can become a character, an environment, or special effect.

A Game Object is a container; you add pieces to the Game Object container to make it into a character, a light, a tree, a sound, or whatever else you would like it to be. Each piece you add is called a component**.**

Creating and Using Scripts

The behavior of GameObjects is controlled by the **Components** that are attached to them. Although Unity's built-in Components can be very versatile, you will soon find you need to go beyond what they can provide to implement your own gameplay features. Unity allows you to create your own Components using **scripts**. These allow you to trigger game events, modify Component properties over time and respond to user input in any way you like.

Unity supports two programming languages natively:

- **C#** (pronounced C-sharp), an industry-standard language similar to Java or C++;
- **UnityScript**, a language designed specifically for use with Unity and modelled after JavaScript;

  In addition to these, many other .NET languages can be used with Unity if they can compile a compatible DLL.

Creating Scripts

Unlike most other assets, scripts are usually created within Unity directly. You can create a new script from the Create menu at the top left of the Project panel or by selecting **Assets > Create > C#**

Script (or JavaScript) from the main menu.

The new script will be created in whichever folder you have selected in the Project panel. The new script file's name will be selected, prompting you to enter a new name.

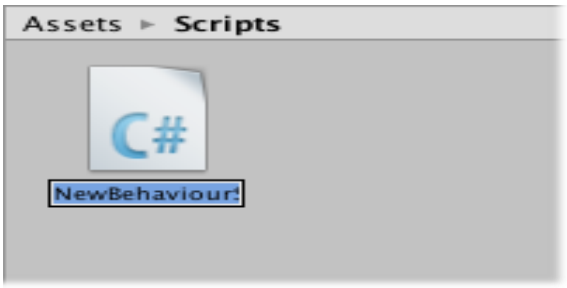


Fig3. *Screenshot For Script Folder*

Anatomy of a Script file

When you double-click a script asset in Unity, it will be opened in a text editor. By default, Unity will use MonoDevelop, but you can select any editor you like from the External Tools panel in Unity's preferences.

The initial contents of the file will look something like this:

```
using UnityEngine;
using System.Collections;
public class MainPlayer : MonoBehaviour
{
   // Use this for initialization
   void Start () {
   }
   // Update is called once per frame
   void Update () {
   }
}
```

A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called **MonoBehavior**. You can think of a class as a kind of blueprint for creating a new Component type that can be attached to GameObjects. Each time you attach a script component to a GameObject, it creates a new instance of the object defined by the blueprint. The name of the class is taken from the name you supplied when the file was created. The class name and file name must be the same to enable the script component to be attached to a GameObject.

The main things to note, however, are the two functions defined inside the class. The **Update** function is the place to put code that will handle the frame update for

the GameObject. This might include movement, triggering actions and responding to user input, basically anything that needs to be handled over time during gameplay. To enable the Update function to do its work, it is often useful to be able to set up variables, read preferences and make connections with other GameObjects before any game action takes place. The **Start** function will be called by Unity before gameplay begins (i.e., before the Update function is called for the first time) and is an ideal place to do any initialization.

Note to experienced programmers: you may be surprised that initialization of an object is not done using a constructor function. This is because the construction of objects is handled by the editor and does not take place at the start of gameplay as you might expect. If you attempt to define a constructor for a script component, it will interfere with the normal operation of Unity and can cause major problems with the project.
A UnityScript script works a bit differently to C# script:

```
#pragma strict
function Start () {

}
function Update () {
}
```

Here, the Start and Update functions have the same meaning but the class is not explicitly declared. The script itself is assumed to define the class; it will implicitly derive from MonoBehavior and take its name from the filename of the script asset.

Controlling a GameObject

As noted above, a script only defines a blueprint for a Component and so none of its code will be activated until an instance of the script is attached to a GameObject. You can attach a script by dragging the script asset to a GameObject in the hierarchy panel or to the inspector of the GameObject that is currently selected. There is also a Scripts submenu on the Component menu which will contain all the scripts available in the project, including those you have created yourself. The script instance looks much like any other Component in the Inspector:

Once attached, the script will start working when you press Play and run the game. You can check this by adding the following code in the Start function:-

```
// Use this for initialization
void Start () {
   Debug.Log("I am alive!");
}
```

Debug.Log is a simple command that just prints a message to Unity's console output. If you press Play now, you should see the message at the bottom of the main Unity editor window and in the Console window (menu: **Window > Console**).

MAIN FUNCTIONS IN GAME PLAY CAMERAS

Cameras are the devices that capture and display the world to the player. By customizing and manipulating cameras, you can make the presentation of your game truly unique. You can have an unlimited number of cameras in a scene. They can be set to render in any order, at any place on the screen, or only certain parts of the screen.

Fig4. *Basic Parameters For Camera*

Properties

Property:	***Function:***
Clear Flags	Determines which parts of the screen will be cleared. This is handy when using multiple Cameras to draw different game elements.
Background	The color applied to the remaining screen after all elements in view have been drawn and there is no skybox.
Culling Mask	Includes or omits layers of objects to be rendered by the Camera. Assigns layers to your objects in the Inspector.
Projection	Toggles the camera's capability to simulate perspective.
Perspective	Camera will render objects with perspective intact.
Orthographic	Camera will render objects uniformly, with no sense of perspective. **NOTE:** Deferred rendering is not supported in Orthographic mode. Forward rendering is always used.
Size (when Orthographic is selected)	The viewport size of the Camera when set to Orthographic.
Field of view (when Perspective is selected)	The width of the Camera's view angle, measured in degrees along the local Y axis.
Clipping Planes	Distances from the camera to start and stop rendering.
Near	The closest point relative to the camera that drawing will occur.
Far	The furthest point relative to the camera that drawing will occur.
Viewport Rect	Four values that indicate where on the screen this camera view will be drawn. Measured in Viewport Coordinates (values 0–1).
X	The beginning horizontal position that the camera view will be drawn.
Y	The beginning vertical position that the camera view will be drawn.
W (Width)	Width of the camera output on the screen.
H (Height)	Height of the camera output on the screen.
Depth	The camera's position

Property:	*Function:*
	in the draw order. Cameras with a larger value will be drawn on top of cameras with a smaller value.
Rendering Path	Options for defining what rendering methods will be used by the camera.
Use Player Settings	This camera will use whichever Rendering Path is set in the Player Settings.
Vertex Lit	All objects rendered by this camera will be rendered as Vertex-Lit objects.
Forward	All objects will be rendered with one pass per material.
Deferred Lighting	All objects will be drawn once without lighting, then lighting of all objects will be rendered together at the end of the render queue. **NOTE:** *If the camera's projection mode is set to Orthographic, this value is overridden, and the camera will always use Forward rendering.*
Target Texture	Reference to a Render Texture that will contain the output of the Camera view. Setting this reference will disable this Camera's capability to render to the screen.
HDR	Enables High Dynamic Range rendering for this camera.
Target Display	Defines which external device to render to. Between 1 and 8.

Details

Cameras are essential for displaying your game to the player. They can be customized, scripted, or parented to achieve just about any kind of effect imaginable. For a puzzle game, you might keep the Camera static for a full view of the puzzle. For a first-person shooter, you would parent the Camera to the player character, and place it at the character's eye level. For a racing game, you'd probably have the Camera follow your player's vehicle.

You can create multiple Cameras and assign each one to a different **Depth**. Cameras are drawn from low **Depth** to high **Depth**. In other words, a Camera with a **Depth** of 2 will be drawn on top of a Camera with a depth of 1. You can adjust the values of the **Normalized View Port Rectangle** property to resize and position the Camera's view onscreen. This can create multiple mini-views like missile cams, map views, rear-view mirrors, etc.

Render path

Unity supports different rendering paths. You should choose which one you use depending on your game content and target platform / hardware. Different rendering paths have different features and performance characteristics that mostly affect lights and shadows. The rendering path used by your project is chosen in **Player Settings**. Additionally, you can override it for each Camera.

Clear Flags
Each Camera stores color and depth information when it renders its view. The portions of the screen that are not drawn in are empty, and will display the skybox by default. When you are using multiple Cameras, each one stores its own color and depth information in buffers, accumulating more data as each Camera renders. As any particular Camera in your scene renders its view, you can set the **Clear Flags** to clear different collections of the buffer information. To do this, choose one of the following four options:

Skybox
This is the default setting. Any empty portions of the screen will display the current Camera's skybox. If the current Camera has no skybox set, it will default to the skybox chosen in the Lighting Window (menu: **Window > Lighting**). It will then fall back to the **Background Color**. Alternatively a Skybox component can be added to the camera. If you want to create a new Skybox, you can use this guide.

Solid color
Any empty portions of the screen will display the current Camera's **Background Color**.

Depth only
If you want to draw a player's gun without letting it get clipped inside the environment, set one Camera at **Depth** 0 to draw the environment, and another Camera at **Depth** 1 to draw the weapon alone. Set the weapon Camera's **Clear Flags** to **depth only**. This will keep the graphical display of the environment on the screen, but discard all information about where each object exists in 3-D space. When the gun is drawn, the opaque parts will completely cover anything drawn, regardless of how close the gun is to the wall.

AUDIO

Fig5. *Screenshot For Audio Recording*

Unity's Audio features include full 3D spatial sound, real-time mixing and mastering, hierarchies of mixers, snapshots, predefined effects and much more.

Basic Theory
In real life, sounds are emitted by objects and heard by listeners. The way a sound is perceived depends on a number of factors. A listener can tell roughly which direction a sound is coming from and may also get some sense of its distance from its loudness and quality. A fast-moving sound source (like a falling bomb or a passing police car) will change in pitch as it moves as a result of the Doppler Effect. Also, the surroundings will affect the way sound is reflected, so a voice inside a cave will have an echo but the same voice in the open air will not.

Fig6. *Audio Recording Mechanism In Unity*

Audio Sources and Listener
To simulate the effects of position, Unity requires sounds to originate from **Audio Sources** attached to objects. The sounds emitted are then picked up by an **Audio Listener** attached to another object, most often the main camera. Unity can then simulate the effects of a source's distance

and position from the listener object and play them to the user accordingly. The relative speed of the source and listener objects can also be used to simulate the Doppler Effect for added realism.

Unity can't calculate echoes purely from scene geometry but you can simulate them by adding **Audio Filters** to objects. For example, you could apply the Echo filter to a sound that is supposed to be coming from inside a cave. In situations where objects can move in and out of a place with a strong echo, you can add a **Reverb Zone** to the scene. For example, your game might involve cars driving through a tunnel. If you place a reverb zone inside the tunnel then the cars' engine sounds will start to echo as they enter and the echo will die down as they emerge from the other side.

The Unity **Audio Mixer** allows you to mix various audio sources, apply effects to them, and perform mastering.
The manual pages for Audio Source, Audio Listener, Audio Mixer, the audio effects and Reverb Zones give more information about the many options and parameters available for getting effects just right.

Working with Audio Assets
Unity can import audio files in **AIFF**, **WAV**, **MP3** and **Ogg** formats in the same way as other assets, simply by dragging the files into the Project panel. Importing an audio file creates an Audio Clip which can then be dragged to an Audio Source or used from a script. The Audio Clip reference page has more details about the import options available for audio files.

For music, Unity also supports tracker modules, which use short audio samples as "instruments" that are then arranged to play tunes. Tracker modules can be imported from **.xm**, **.mod**, **.it**, and **.s3m** files but are otherwise used in much the same way as ordinary audio clips.

Audio Recording
Unity can access the computer's microphones from a script and create Audio Clips by direct recording. The **Microphone** class provides a straightforward API to find available microphones, query their capabilities and start and end a recording session. The script reference page for Microphone has further information and code samples for audio recording.

Supported formats

Format	Extensions
MPEG layer 3	.mp3
Ogg Vorbis	.ogg
Microsoft Wave	.wav
Audio Interchange File Format	.aiff / .aif
Ultimate Soundtracker SSmodule	.mod
Impulse Tracker module	.it
Scream Tracker module	.s3m
FastTracker 2 module	.xm

Canvas
The **Canvas** is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.
Creating a new UI element, such as an Image using the menu **GameObject > UI > Image**, automatically creates a Canvas, if there isn't already a Canvas in the scene. The UI element is created as a child to this Canvas.
The Canvas area is shown as a rectangle in the Scene View. This makes it easy to

position UI elements without needing to have the Game View visible at all times.
Canvas uses the EventSystem object to help the Messaging System.

Render Modes
The Canvas has a **Render Mode** setting which can be used to make it render in screen space or world space.
Screen Space - Overlay
This render mode places UI elements on the screen rendered on top of the scene. If the screen is resized or changes resolution, the Canvas will automatically change size to match this.

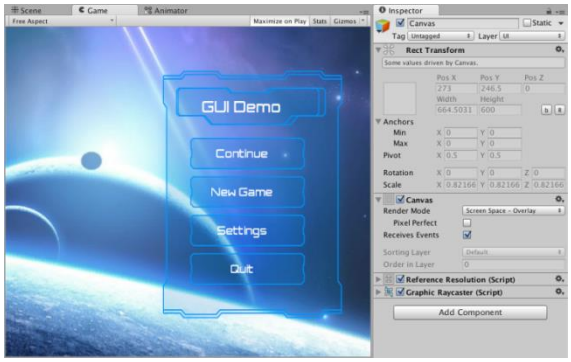


Fig7.*Ui In Screen Space Overlay Canvas*

Anaylsis Of Game With Existing Systems
Comparative Analysis
The morality system in this game is usually hailed as unique and complex because of its interdependent object system. In closer observation, we found that many other games are available in market with same functionalities. In our development phase, *Age of Empire* and *Clash of clan* are at our target. Many things like time management, object depends on each other, score rate, buildings drag and drop functionality etc. are similar things that we have developed according to these existing game. There is major difference in between *Innovating India* and *Age of empire, Clash of Clan* is that, these games provide a fixed focus camera mechanism in game but we have provided a first person controller in game which quite gives real experience of play to the user. Another difference is that; *Age of Empire* provides a different timeline and game story like game started from dark age and ended in modern age but we have provided a village development up to the smart city development.

Interpretation of Game
This type of analysis gives exact information about what happens in the game. When we observe game *Innovating India* from particular standpoint, Game shows village development extends up to the modern smart city. In *Innovating India* player has to collect coins, woods, foods and then try to build a village by dragging and dropping buildings. Each time score will be updated. New building requires more resources and player has to collect more and more resources on terrain and game continues. In this way, new buildings gets available to player as he/she moves in game and game ends with a modern smart city.

Historical Analysis
As we have analyzed recent past of gaming industries for development and public interest in playing game; is somewhat changing. The action, shooting games are being avoided by public and they are more attracted towards the complex and strategic games. The *Age of Empire, 2002* worked very impressively for understanding human development through game play in past decade. *Clash of Clan, 2013* also the named as one of the best android based strategic game with 50 million downloads. Both games did well and still doing well in game industry. Both games were using same game design, functionalities, and game play mechanism. They are the best strategic game for decades.

ALGORITHMS IN GAMEPLAY

Crash and Turn

Let's start at point A, to find a way to point B.

Try to move in a straight line as long as we can.When an obstacle appears, choose one of the two sides (left or right).Now go around it using the left- or right-hand rule-follow the object parallel to its sides until we have open line of sight of the destination again and thus can return to the straight line of advance.

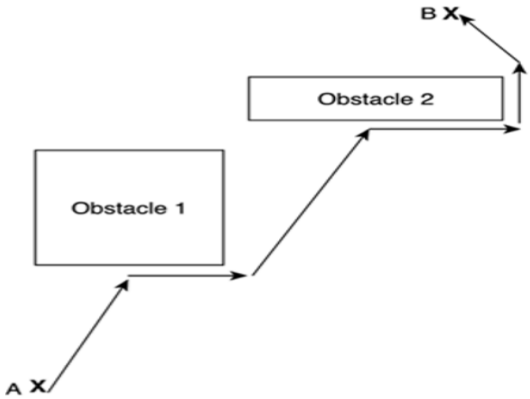


Fig 8. *Crash And Turn Algorithm*

Two possibilities are

1. Choosing the side that deviates less from the initial trajectory
2. Choosing a random side

Crash and turn always finds a way from origin to destination if we can guarantee that obstacles are all convex and not connected. The algorithm is quite lightweight and thus can be implemented with very low CPU impact.

A* Algorithm

The most popular algorithm for this problem is called A* and basically builds the graph at runtime, searching for the shortest path from an initial state to the end state. We need an algorithm that somehow understands the difference between a good path and a bad path, and only examines the best candidates, forgetting about the rest of the

- Let's Start with the base node, expand nodes using the valid moves.

$$f(node) = g(node) + h(node)$$

- Where f(node) is the total score we assign to a node.
- For now, suffice it to say that g(node) is the portion that takes the past decisions into consideration and estimates the cost of the path we have already traversed in moves to reach the current state.
- The h(node) is the heuristic part that estimates the future. Thus, it should give an approximation of the number of moves we still need to make to reach our destination from the current position.

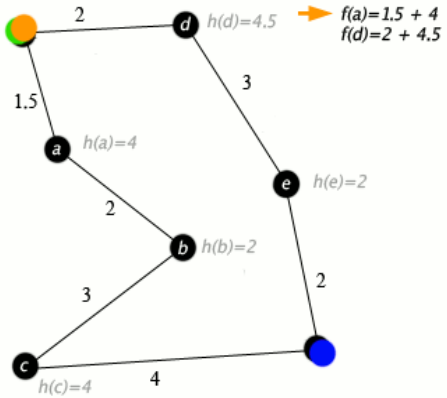


Fig9. *A* Algorithm*

CONCLUSION

This paper gave an information about how to develop game in Unity game engine. Different key components of the assets were studied and with the help of those components different kind of assets were designed in Maya Autodesk. Throughout the development process of this game we have learned many basic functionalities of Unity game engine like camera settings, UI elements and audio recording of scenes as mentioned above. This paper briefs an information about basic core algorithms used in development of game play like crash & turn and A* algorithm.

REFERENCES

1. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.435.1865&rep=rep1&type=pdf

2. http://www.ijceronline.com/papers/Vol2_issue4/P02410531059.pdf
3. http://yannakakis.net/wp-content/uploads/2012/02/PhDThesis.pdf
4. http://users.csc.calpoly.edu/~fkurfess/480/F04/Misc/AI-Tools-Games.pdf
5. https://knowledge.autodesk.com/support/maya
6. https://docs.unity3d.com/Manual/index.html