SVGAssets 1.3.5 (AmanithSVG for Unity)

Matteo Muratori, Michele Fabbri

SVGAssets, a Unity native plugin for reading SVG files and render them on Texture2D objects, at runtime. Technical description, limitations and requirements.

Mazatech S.r.l.
Via Livatino, 2
47020 – Longiano (FC) – Italy
info@mazatech.com
eMail: info@mazatech.com
May 05, 2015

1. Description

SVGAssets plugin consists of a native library called AmanithSVG and an object-oriented C# interface to the low level library.

SVGAssets simplifies the management of image assets for all those projects that need to run across a wide range of devices with different resolutions: using the plugin, the developer keeps all image files in SVG format and render them on the target platform at runtime, at the desired resolution.

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C) since 1999.

The C# interface exposes classes for the manipulation of colors, viewports, drawing surfaces and SVG documents: there are methods for creating drawing surfaces, load SVG documents, get their information and draw them on surfaces.

The Document Object Model (DOM) is not exposed because the pluging is aimed to the rendering only.

Some headlines about the plugin:

- Based on the robust AmanithVG SRE software rendering engine
- Really high antialiasing quality: analytical pixel coverage
- Really fast rendering: 30ms tiger on Nexus4 at max resolution
- Support of SVG Tiny 1.2 specifications with the exception of animation, text, images
- Support of some SVG Full 1.1 features: radial gradients, gradients spread modes, path 'd' attribute complete syntax, inline styles.
- Cross platform: desktop (Win, OSX, Linux), mobile (iOS7+, Android), Windows Phone 8/8.1

The package includes:

- binary distribution of the AmanithSVG library for mobile and desktop platforms
- a standalone SVG player based on the SVGAssets technology, for desktop platforms; this player allows you to test SVG assets before to use them inside your Unity projects
- Unity editor scripts that automate the generation of texture atlases and sprites from SVG files
- Unity build scripts that automate the deployment in a way such that texture atlases and sprites will be generated on the device at initialization/runtime, instead to be included as heavy bitmap files
- a couple of Unity example scenes, showing the plugin usage

2. Package content

SVGAssets package consists of the following “Assets” subfolders:

“Anim”: animation files relative to the orc example scene

“Atlases”: texture atlases and sprites generated by SVGAssets

“Editor”: Unity editor scripts

“Plugins”: AmanithSVG native library for all supported platforms

“Scenes”: example scenes

“Scripts”: scripts for the manipulation of colors, viewports, drawing surfaces, SVG documents, sprites

“SVGFiles”: all used SVG

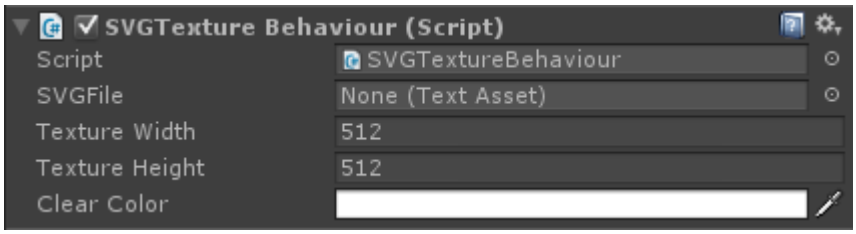
“Tools”: a standalone SVG player based on the SVGAssets technology, for desktop platforms

3. From SVG to texture: the scripts

In addition to the native library and its low level C# interface, SVGAssets package includes some scripts to simplify the workflow in Unity:

SVGTextureBehaviour

This monobehaviour script performs a single SVG file rendering on a Texture2D.

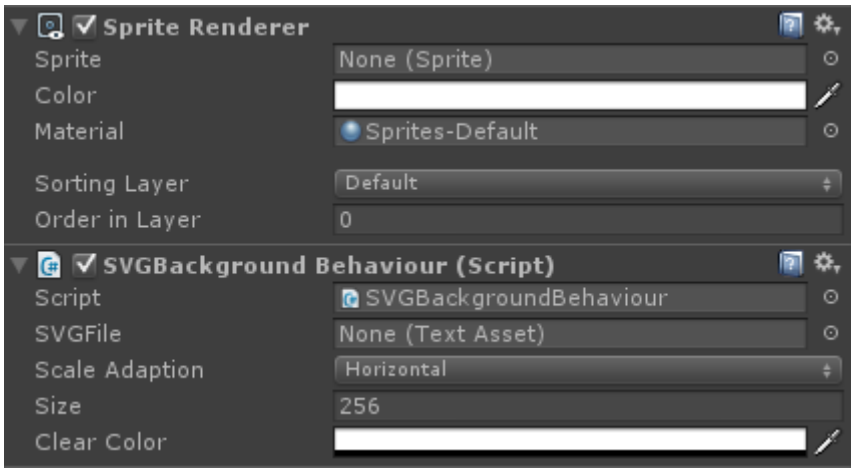


SVGTextureBehaviour script interface

It takes in input the SVG file (as a TextAsset), the texture dimensions and a clear color. Such color will be used to clear the surface before to start the rendering. The rendering is performed in the Start function, that assigns the generated texture to the "renderer.material.mainTexture" field of the GameObject.

SVGBackgroundBehaviour

This monobehaviour script performs the rendering of a single SVG file on a Texture2D, and creates a new sprite out of it (covering the whole texture). The sprite is then assigned to the SpriteRenderer component.



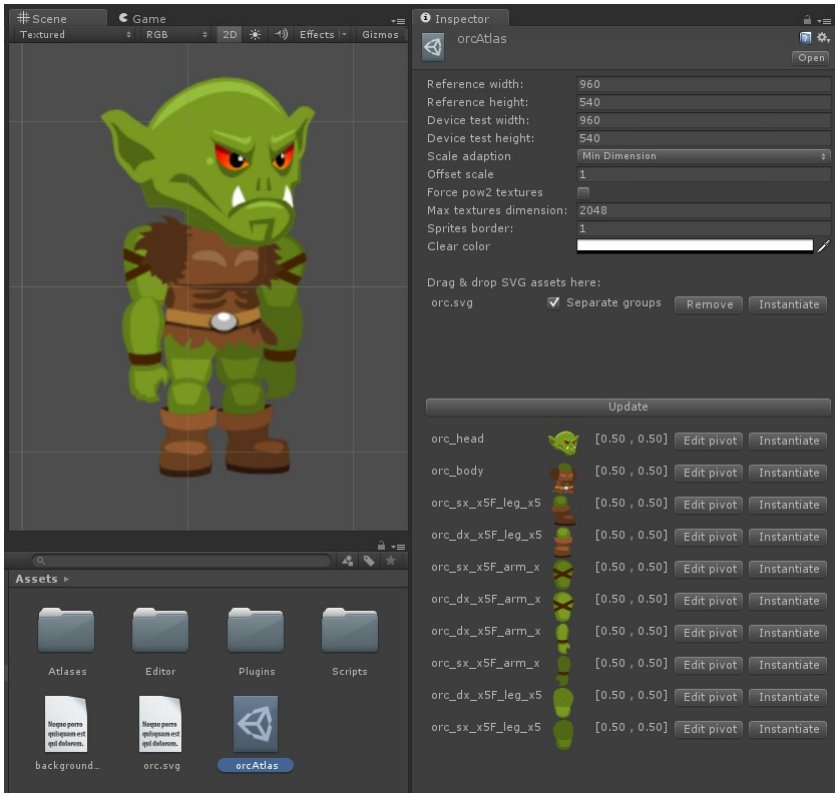
SVGBackgroundBehaviour script interface

The rendering is performed keeping the size (horizontal or vertical) equal to the specified value (in pixels), and calculates the other dimension preserving the original SVG aspect ratio. As for the SVGTextureBehaviour script, it is possible to specify a clear color too. The size parameter can be changed at runtime: at the next Update monobehaviour call, the rendering will be updated.

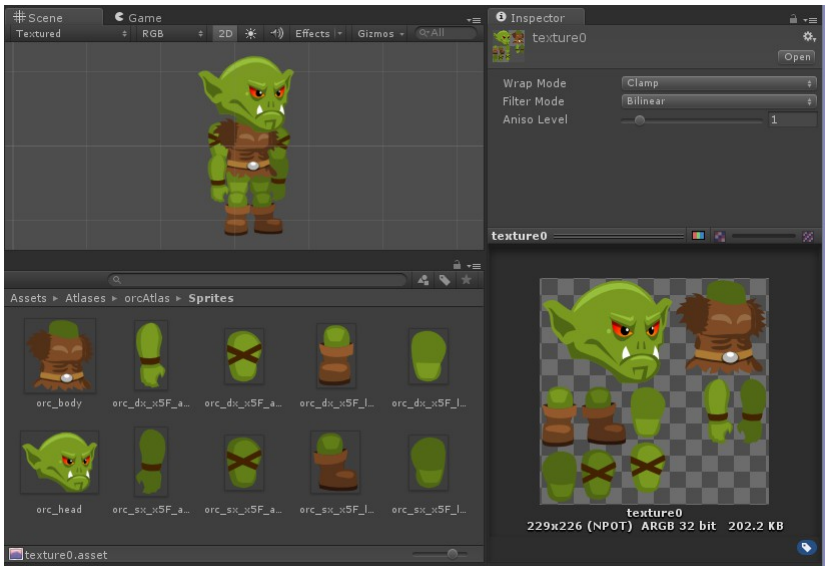
NB: in order to work properly, this script must be assigned to a Sprite object.

SVGAtlas

This ScriptableObject allows the generation of sprites from different SVG files, packing the generated sprites into one or more textures atlas, according to the specified parameters.



An SVGAtlas asset, with groups rendered as sprites



A generated texture atlas, along with packed sprites

Each SVG file can be drag&dropped in the relative section (the region labeled with “Drag & drop SVG assets here”). Dragged items can be moved in the list, changing their order: this will induce an automatic z-order (i.e. “Order in Layer” value of SpriteRenderer components) of generated sprites. In the detail, sprites will be ordered in a way such that the one on the top of the list will be placed behind all others.

For each SVG entry, it is possible to specify if the altas generator must render it as a whole single sprite (“Separate groups” option unchecked), or if first level groups (<g> tags) must be rendered as separate sprites (“Separate groups” option checked). This option will allow the animation of each group.

Being vector based, SVG can be freely scaled without visual quality loss. In order to give the user an effective way to control the scaling factor, SVGAssets makes available the following parameters:

- Reference width, height: it's the resolution at which SVG files would be rendered at the native dimensions, specified within the SVG file itself (outermost <svg> element, attributes “width” and “height”). It is really important that all used SVG files contain such attributes, because if they are not available, SVGAssets will render each sprite filling a whole texture.
- Device test (simulated) width, height: it's the resolution that will be simulated to generate and test sprites in the editor. These settings allow the user to test different device resolutions in advance.
- Scale adaption: it defines which device dimension will control the scaling factor.
- Offset scale: an additional scale factor that will be applied to all SVG files.

The general formula to calculate the scaling factor is:

Scl = (Device dimension / Reference dimension) * OffsetScale

where “Device dimension” is the one chosen by the ScaleAdaption parameter.

For example, lets have:

- an SVG file with 100 x 150 dimensions (i.e. <svg width=”100px” height=”150px”>)
- a reference resolution set to 1024 x 768
- scale adaption set to Vertical
- OffsetScale set to 1

On a 640 x 480 device resolution, the sprite will be rendered at 62 x 94 pixels, because scaling factor is 0.625 (480 / 768).

On a 1536 x 2048 device resolution, the sprite will be rendered at 266 x 400 pixels, because scaling factor is 2.667 (2048 / 768).

Two script parameters control some aspects relative to the generated textures:

- Force pow2 textures: if checked, this parameters will force the atlas generator to produce textures whose dimensions are power-of-two values.
- Max textures dimension: the maximum dimension allowed during the textures atlas generation. Please make sure that such value won't exceed the real device capability.

The last two script parameters control how many pixels will separate each sprite from any other (“Sprite border”) and the color that is used to clear the surface before to start the rendering (“Clear color”).

SVGAtlasEditor

This editor script implements the inspector mask for the editing of SVGAtlas assets. It also contains a SVGBuildProcessor static class that will take care of some aspects relative to the building phase. In particular, it ensures that during the building process all generated atlases won't be included in the final package (actually they are substituted by a dummy 1x1 texture), because they will be regenerated at runtime on the device. This will reduce the build size, showing the advantage of using SVG files instead of bitmaps.

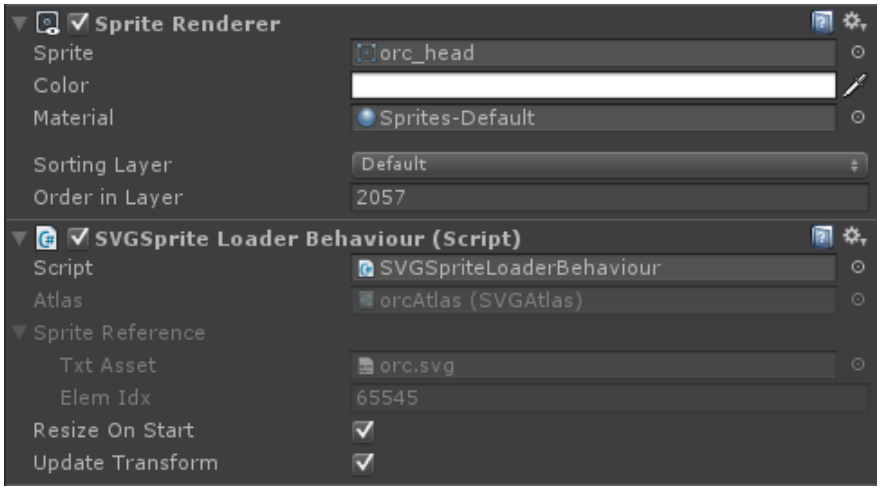
SVGPivotEditor

This editor script allows the user to edit the pivot point of generates sprites. The pivot can be moved by simply clicking the mouse on the desired position. It is desirable to setup pivot points before animating sprites.

The editor of pivot point

SVGSpriteLoaderBehaviour

This monobehaviour script takes care to render the sprite at runtime on the device, according to its original SVGAtlas generator settings.



SVGSpriteLoaderBehaviour component interface

Editable parameters consist in:

- Resize on start: if checked (default value), the sprite will be regenerated at the Start of monobehaviour, else the user must update the sprite programmatically calling the public UpdateSprite function.
- Update transform: if checked (default value), at every monobehaviour LateUpdate call, the sprite local position will be fixed according to its current dimensions. This will ensure that animated child sprites will be in the correct position relative to their father (just the position is relevant, rotation and scale do not need to be fixed). If the sprite is moved programmatically by code (e.g. a root body part of a character), this option must be unchecked.

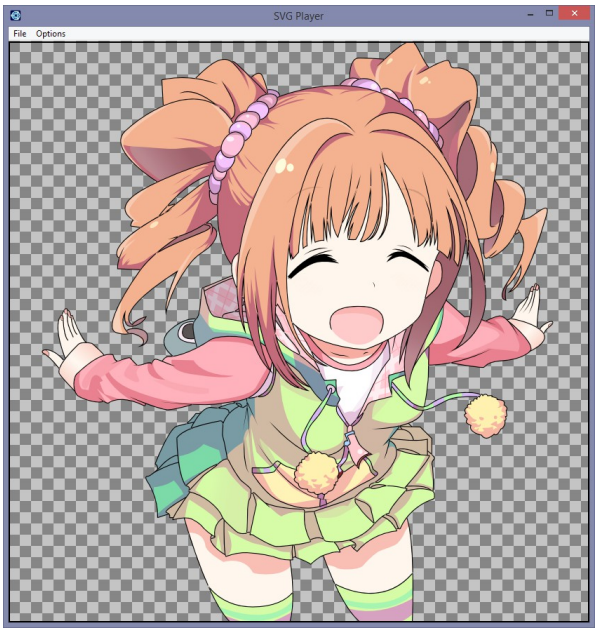
SVGRenamerImporter

This script implements an asset post-processor (AssetPostprocessor class). Its task consists in changing the extension of new asset files from ".svg" to ".svg.txt", so Unity can recognize those files as text assets (see http://docs.unity3d.com/Manual/class-TextAsset.html).

4. Step-by-step usage guide: a minimal example

The example explained in this chapter is contained in the “Scenes/plane.unity” file. The example shows a simple plain with an attached texture generated from an SVG file.

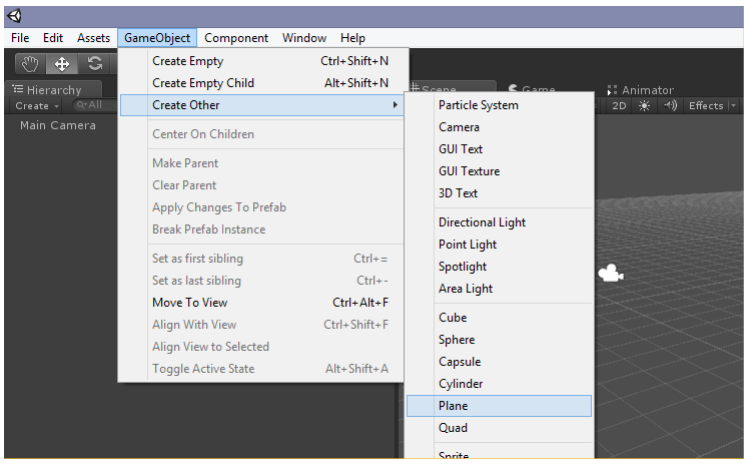
Before to start the Unity IDE, it's a good practice to test your SVG files using the included SVG player, in order to verify that the rendering is consistent with the output of your SVG editor. Take into account that SVGAssets plugin has some limitations (see Chapter 6).



girl.svg looks good on the standalone player

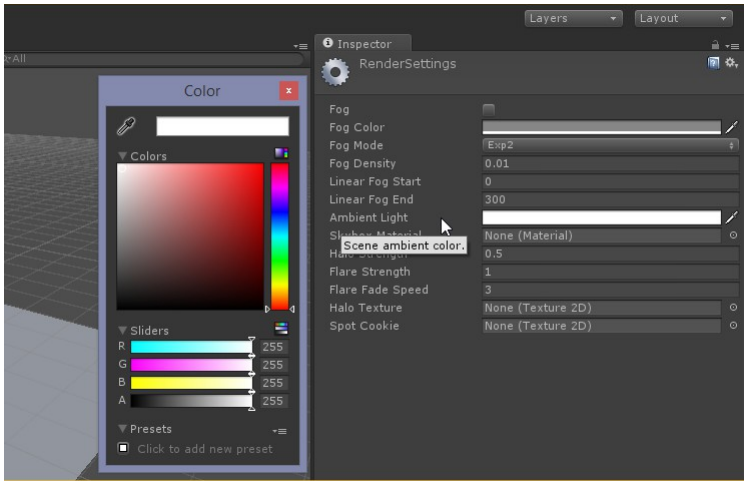
The example has been realized following these steps:

- Create a new Unity project, SVGAssets can be used for 2D and 3D project; in this case we accept the default 3D setup
- Because a project is a new one, it is mandatory to copy the whole Plugins, Editor and Scripts folders inside the project Assets folder; so the native AmanithSVG libraries and its C# interface will be available for the project
- Create a plane object (menu GameObject → Create Other → Plane) and make sure that the main camera is pointing it



The plane object creation

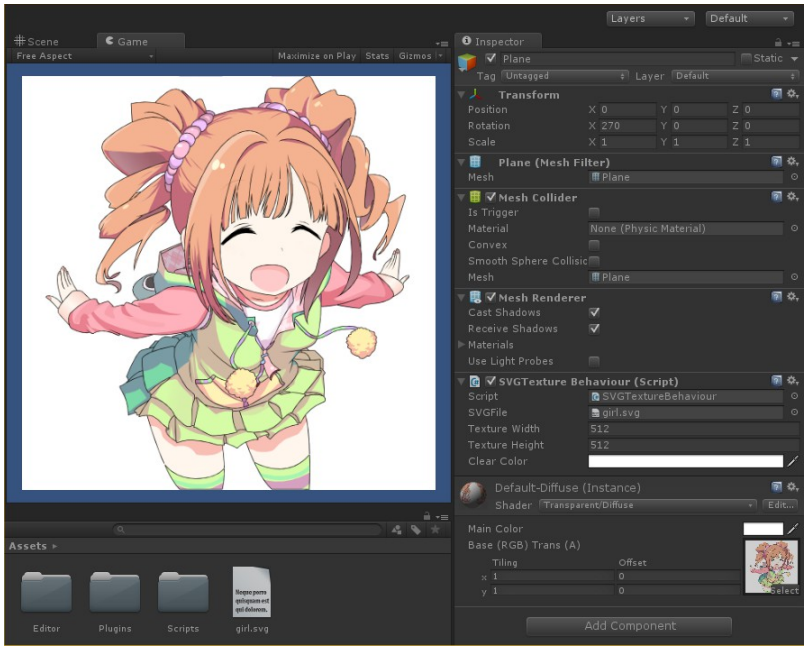
- Set a full white ambient light (menu Edit → Rendering Settings), in order to display SVG files with their original colors



Ambient light settings

- Copy the SVG file (e.g. girl.svg) in the Assets directory. Such file will be renamed automatically by SVGRenamerImporter script, adding an additional txt extension, so Unity can recognize it as a TextAsset.
- Select the plan and attach the SVGTextureBehaviour script (menu Component → Add). This script is a really simple example showing a possible usage of SVGAssets library. In the detail the script:
  - Creates the texture with the specified dimensions
  - Creates a drawing surface with the same dimensions
  - Creates and load the SVG document
  - Renders the SVG document to the drawing surface
  - Uploads the drawing surface pixels to the texture
  - Returns the created texture

- Drag&drop the girl SVG file from the Project window to the SVGFile property field of the script. Let other properties unchanged to their default values (512 x 512 pixels texture dimensions, white background clear color).
- Click play and see the result.



This is the result after pressing Play

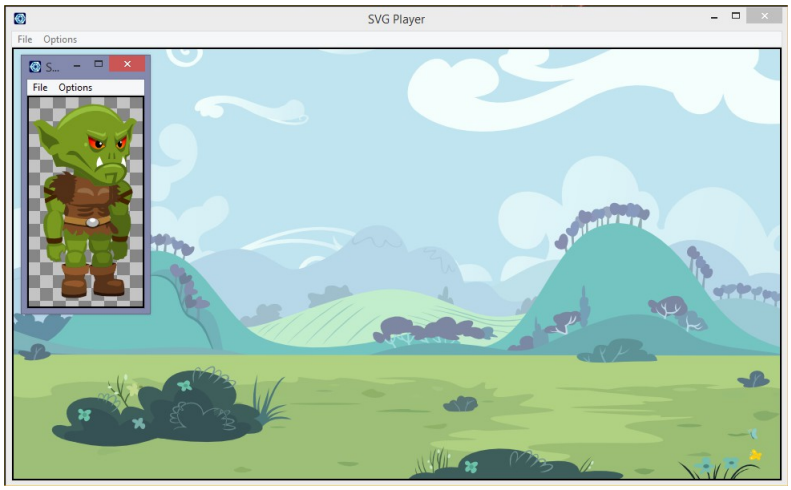
In the package there is also included another script called SVGTextureAtlasBehaviour; in order to test it, stop the current execution and perform the following steps:

- Select the plane and remove the attached SVGTextureBehaviour component
- Add the SVGTextureAtlasBehaviour script component: this script allows you to specify four different SVG files to be rendered on a single texture
- Drag&drop four SVG files from the Project window to SVGFile1, 2, 3, 4 property fields of the script.

As before, click play and see the result in the Game window.

5. Step-by-step usage guide: a 2D game-like example

The example explained in this chapter is contained in the “Scenes/orc.unity” file. The example implements a simple 2D character (the orc), walking on a background. Both orc and background are native SVG files.



Orc and background SVG look good on the standalone player

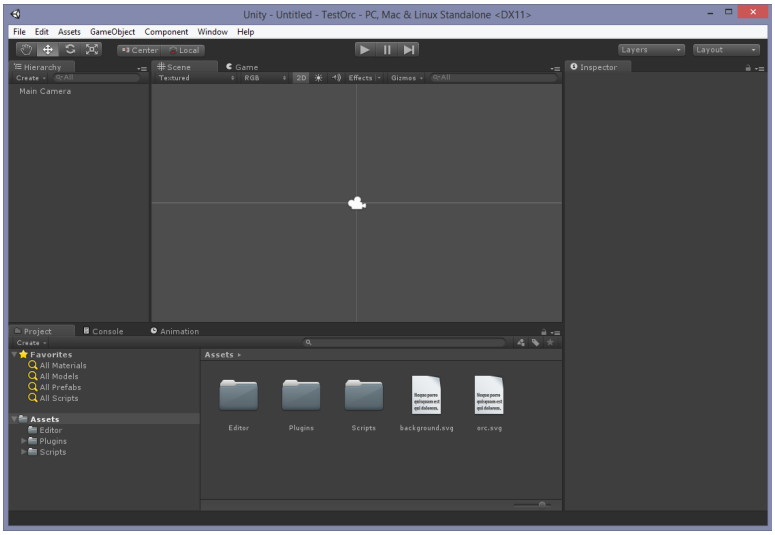
The orc asset is an SVG file where each body part is a separate first level group (<g> tag), each group has a well defined name (id attribute). This setup will allow to render each body part as a separate animable sprite. SVGAssets atlas generator will perform an optimized sprite packing routine in the editor, and later on the target device.

Illustrator allows to show and edit groups

Now we can proceed in the following way:

- Create a new Unity project, choosing a 2D setup
- Because a project is a new one, it is mandatory to copy the whole Plugins, Editor and Scripts folders inside the project Assets folder; so the native AmanithSVG libraries and its C# interface will be available for the project

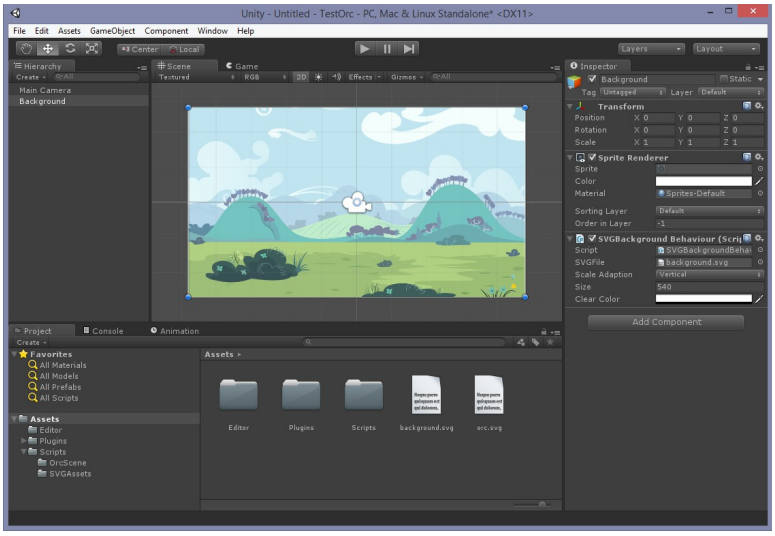
- Copy the Anim folder too (it contains two simple “idle” and “walking” animation assets), inside the project Assets folder
- Copy the orc.svg and background.svg files in the Assets folder. Such files will be renamed automatically by SVGRenamerImporter script, adding an additional txt extension, so Unity can recognize them as a TextAsset



The project layout after copying all needed files

- Create the sprite that we will use as background (menu GameObject → Create Other → Sprite). We rename the created object as Background. Now we can attach a SVGBackgroundBehaviour script to it, then set its fields:
  - drag&drop the background.svg file in the “SVGFile” field
  - set scale adaption to vertical, because we want the background to cover the whole device screen height, making the horizontal direction scrollable
  - set the size to 540, that is the original background.svg vertical size (i.e. “height” attribute)

  In addition we set the “Order in Layer” value of the SpriteRenderer component to -1, so the background sprite will always be behind all other sprites.



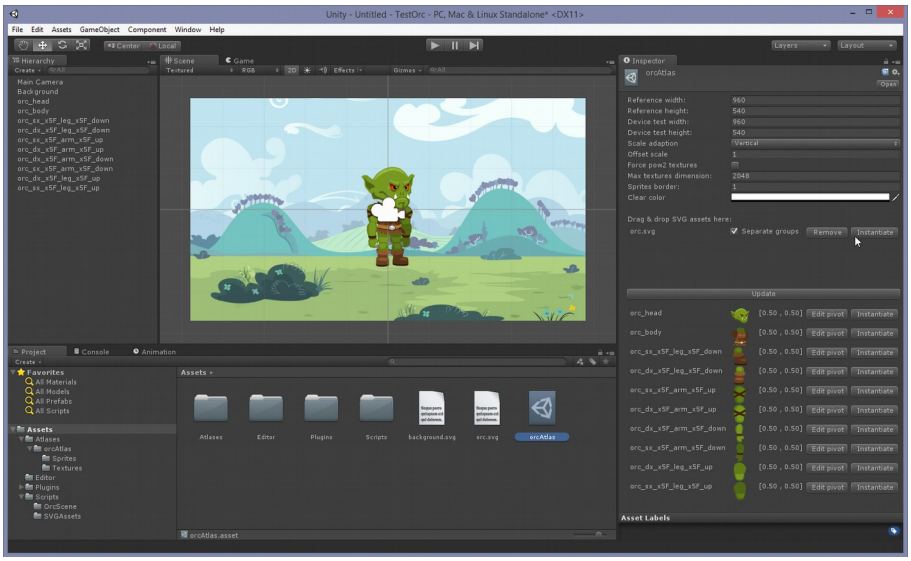
The background SVG placed on the scene

- Create an SVGAtlas atlas generator (menu Assets → Create → SVGAtlas), that will be used to create and pack all the sprites relative to the orc body parts. We rename the created asset in “orcAtlas” just to avoid confusion. Then we can proceed with its settings:
  - drag&drop the orc.svg asset to the region labeled with “Drag & drop SVG assets here” and check the “Separate groups” option
  - set reference width and height to 960 x 540 (the dimensions of the background): at design time, we want to see if the orc character size is “compatible” with the background
  - set device test width and height to 960 x 540, so in the editor we can simulate a device with a resolution equal to the background; setting a device test resolution equal to the reference resolution has the effect to generate all the orc sprites at their original dimension, as specified in the orc.svg document
  - Set scale adaption to vertical, for the same reason that we discussed for the background
  - Click the “Update” button

At this point, the texture atlas and the sprites assets relative to orc body parts have been generated, and made available in the editor. Please note that each sprite starts with a pivot point equal to [0.5, 0.5], that means “the center of sprite”.

The generated orc sprite assets

- Now we can instantiate the whole orc (i.e. all its parts), clicking the “Instantiate” button present in the row where the orc.svg has been dropped. In this way, previously generated sprites assets will be instantiated in the scene.

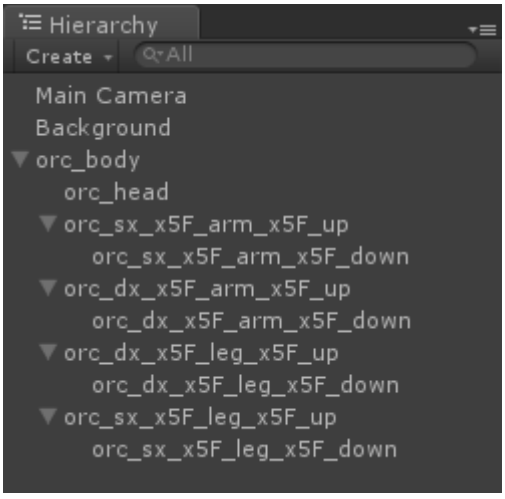


Orc has been instantiated with just one click

Please note that each instantiated sprite:

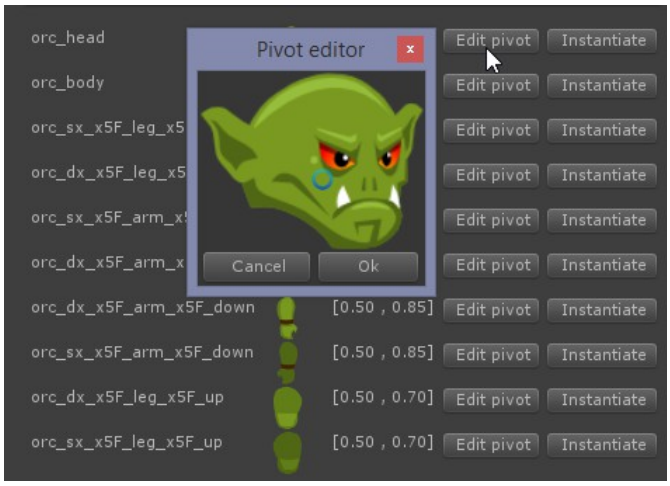
  - has an attached SVGSpriteLoaderBehaviour script component, that will take care to regenerate the sprite on the device, at runtime, with the correct scale
  - has the “Order in Layer” value of the SpriteRenderer component automatically set, in order to respect the correct z-order induced by the sequence in which groups appear in the orc.svg file

- In order to animate each body part like a real character, we must setup:
  - the correct hierarchy of all instantiated sprites (in the Hierarchy section on the left, simply drag&drop each child object under its father): we want the body part to be the root of our hierarchy



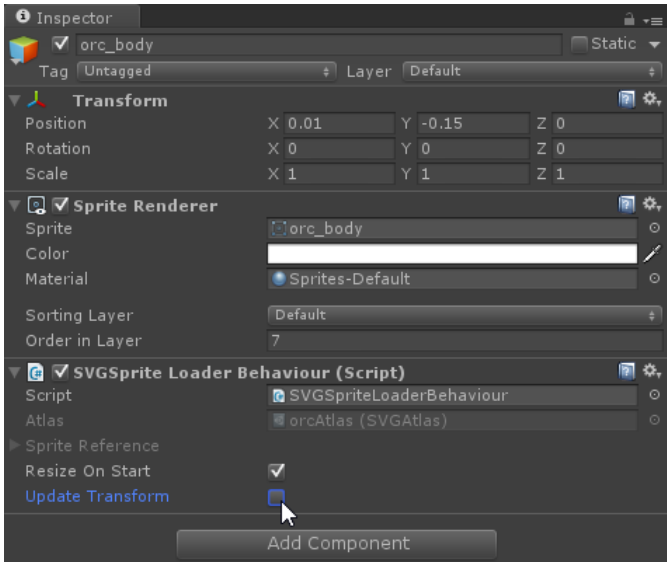
Orc hierarchy, body is the root

  - the pivot points of each sprite: to do this, use the pivot editor made available by the orcAtlas object



The pivot window editor

Because we are going to move the orc (actually the body part) programmatically by code, we must select the "orc_body" object and uncheck the "Update Transform" option of the SVGSpriteLoaderBehaviour component.



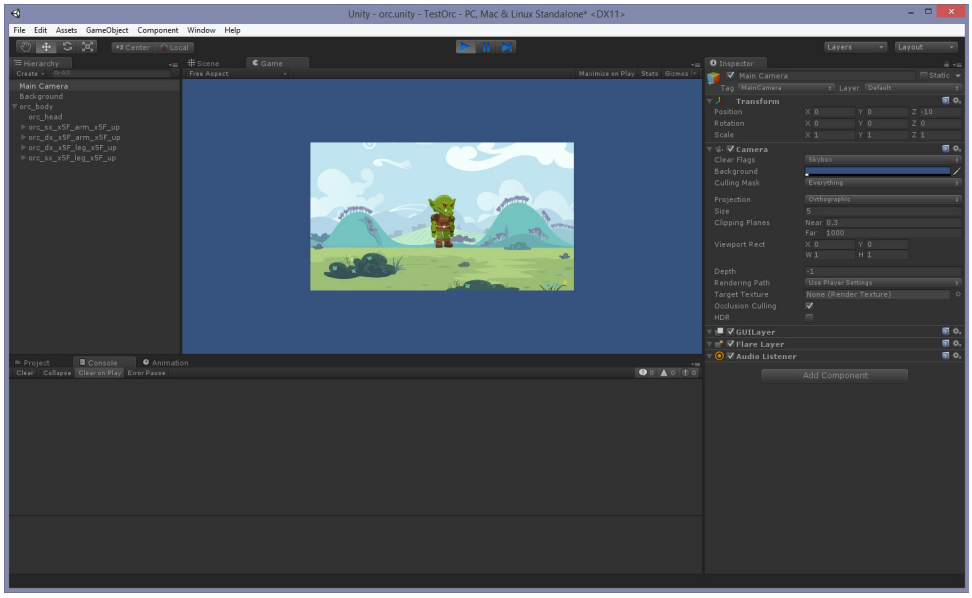
Update Transform must be unchecked for the body part

Now the scene is almost complete, so we can save it.

The scene is now complete

If we click the Play button, we see that:

- orc sprites are resized, but the character position is not consistent respect to the Scene view
- background sprite is not resized
- the main camera viewing volume is not covering the exact height of the background



Playing the scene

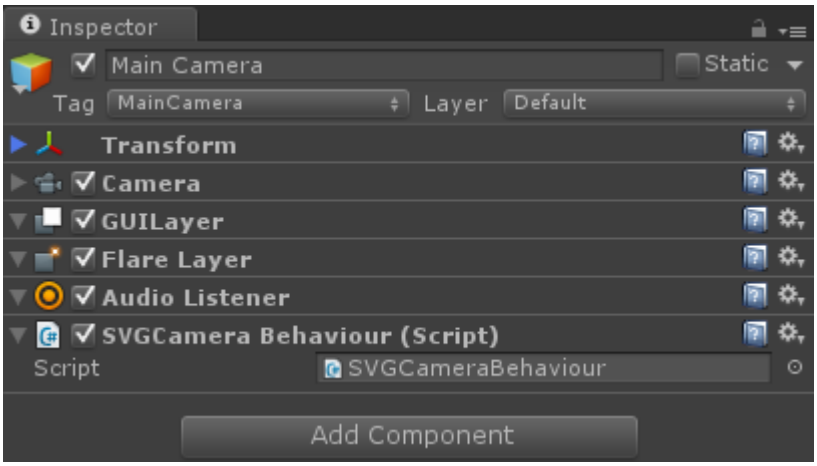
The reason is because at design time, in the editor, all sprites and their positions are valid for a (test) device resolution equal to 960 x 540. So, we want to:

- make the camera viewing volume (actually the vertical size) to cover the whole screen height: this means to set the Camera.orthographicSize properly

- send a resize event to the background: this means to set its SVGBackgroundBehaviour.Size property
- send a resize event to the orc character: this means to call the orc body sprite (actually its SVGSpriteLoaderBehaviour component) UpdateSprite function, specifying to update children too

In this way, when the camera script detects a change in the vertical screen resolution (e.g. due to a device orientation switch), we can resize all sprites automatically.

In order to accomplish these tasks, we add a script to the main camera (SVGCameraBehaviour): this script, at each monobehaviour Update call, checks for a screen resolution change, and if this event occurs, it informs all its listeners:



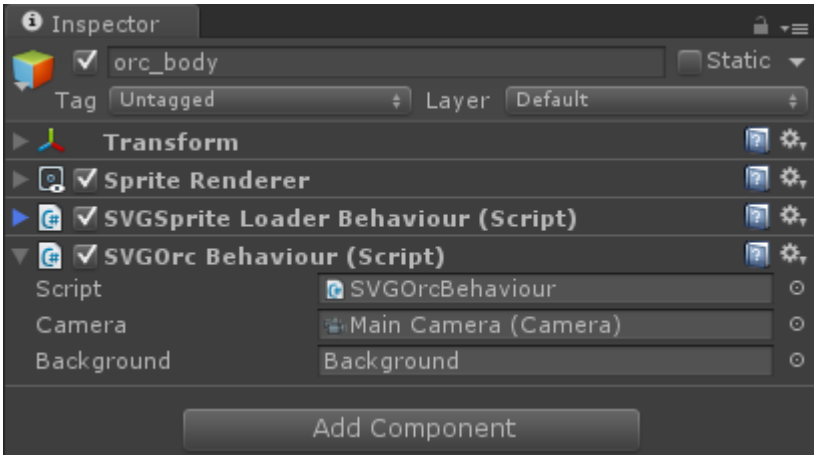
The camera script detects if screen resolution has changed

```
private void Resize(int newScreenWidth, int newScreenHeight) {
    // map the camera rectangle to the whole screen
    this.camera.orthographicSize = (newScreenHeight * this.camera.rect.height) /
                                   (SVGAtlas.SPRITE_PIXELS_PER_UNIT * 2);
    // call OnResize handlers
    this.OnResize(newScreenWidth, newScreenHeight);
}
void Start() {
    // at the first Update() we force a resize event
    this.m_LastScreenHeight = -1;
}
void Update() {
    // get the current screen size
    int curScreenWidth = Screen.width;
    int curScreenHeight = Screen.height;
    // if screen size has changed (e.g. device orientation changed), fire the event
    // NB: in this example we do not need/use the screen width
    if (curScreenHeight != this.m_LastScreenHeight) {
        this.Resize(curScreenWidth, curScreenHeight);
```

```
        // in this example we do not need/use the screen width
        this.m_LastScreenHeight = curScreenHeight;
    }
}
```

At this point, we add a script to the orc body sprite (SVGOrcBehaviour), that has the following input fields:

- the scene camera, so we can implement and register the OnResize event handler
- the background sprite, so we can set its Size property within the OnResize event handler



The orc script, attached to the body sprite

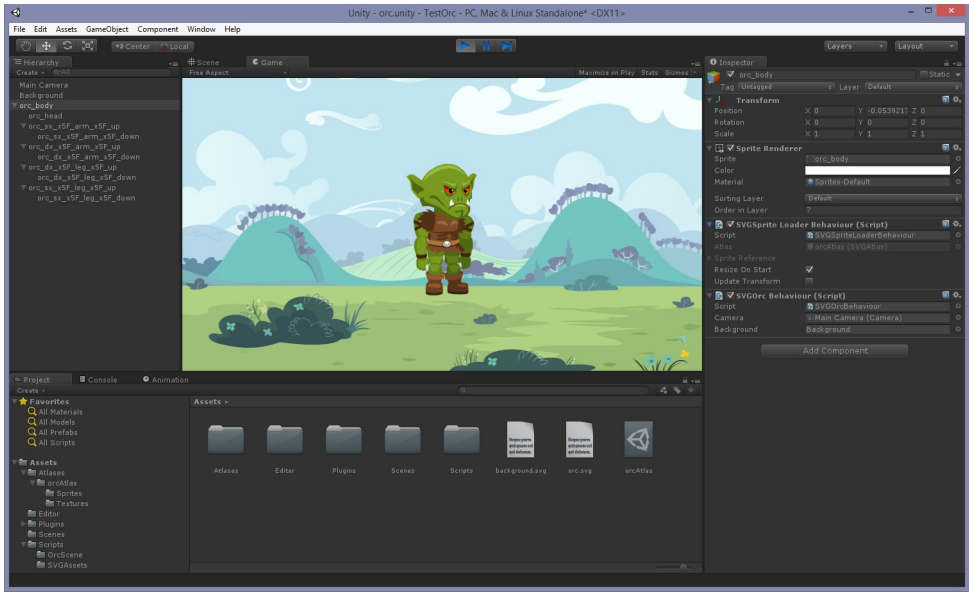
The SVGOrcBehaviour script implements the OnResize handler (called by the camera when screen resolution changes); such handler receives the current screen dimensions and:

- sets the background vertical size
- resizes all orc sprites
- resets the orc character position of the world origin

```
private void CameraPosAssign(Vector3 orcPos) {
    // set the camera according to the orc position
    this.Camera.transform.position = new Vector3(orcPos.x, 0, -10);
}
private void ResetOrcPos() {
    // move the orc at the world origin and set the camera according to the orc position
    Vector3 orcPos = new Vector3(0, 0, 0);
    this.transform.position = orcPos;
    this.CameraPosAssign(orcPos);
}
private void OnResize(int newScreenWidth, int newScreenHeight) {
    // render the background at the right (vertical) resolution
    this.m_Background.Size = newScreenHeight;
```

```
    // get the orc (body) sprite loader
    SVGSpriteLoaderBehaviour spriteLoader = gameObject.GetComponent<SVGSpriteLoaderBehaviour>();
    // update all sprites that form the orc
    spriteLoader.UpdateSprite(true, newScreenWidth, newScreenHeight);
    // move the orc at the world origin and set the camera accordingly
    this.ResetOrcPos();
}
void Start() {
    this.m_Background = this.Background.GetComponent<SVGBackgroundBehaviour>();
    // register handler for device orientation change
    this.m_Camera = this.Camera.GetComponent<SVGCameraBehaviour>();
    this.m_Camera.OnResize += this.OnResize;
}
```

Now if we click play, we see that the camera viewing volume is covering the whole screen height, the background sprite is resized according to the camera pixels height, the orc character has the same vertical proportion respect to the background.



The next step is to move the orc along the horizontal axis, and enable the camera to follow it and scroll the background. To accomplish these tasks, the SVGOrcBehaviour script:

- sets the background position to the world origin (0,0, 0), so the background won't move
- implements two functions that move the orc left or right, changing its body world position (the y coordinate will be fixed); such functions will be called in the LateUpdate routine, if the user has pressed the mouse button
- implements a function that derives the camera position according to the orc position

```
private void Move(Vector3 delta) {
    // move the orc
```

```
    Vector3 orcPos = this.transform.position + delta;
    // orc body pivot is located at 50% of the whole orc sprite, so we can calculate bounds easily
    float orcWorldLeft = orcPos.x - this.m_WorldWidth / 2;
    float orcWorldRight = orcPos.x + this.m_WorldWidth / 2;

    if (orcWorldLeft < this.m_Background.WorldLeft)
        orcPos.x += (this.m_Background.WorldLeft - orcWorldLeft);
    else
    if (orcWorldRight > this.m_Background.WorldRight)
        orcPos.x -= (orcWorldRight - this.m_Background.WorldRight);

    this.transform.position = orcPos;
    // set the camera according to the orc position
    this.CameraPosAssign(orcPos);
}
private void MoveLeft() {
    // flip the orc
    this.transform.localScale = new Vector3(-1, this.transform.localScale.y,
                                            this.transform.localScale.z);
    this.Move(new Vector3(-WALKING_SPEED, 0, 0));
}
private void MoveRight() {
    this.transform.localScale = new Vector3(1, this.transform.localScale.y,
                                            this.transform.localScale.z);
    this.Move(new Vector3(WALKING_SPEED, 0, 0));
}
void LateUpdate() {
    if (Input.GetButton("Fire1")) {
        Vector3 worldMousePos = this.Camera.ScreenToWorldPoint(Input.mousePosition);
        if (worldMousePos.x > this.transform.position.x)
            this.MoveRight();
        else
        if (worldMousePos.x < this.transform.position.x)
            this.MoveLeft();
    }
}
```

All the code (orc and camera movement) is based on world coordinates. In order to pass from pixel units to world units, it's simply a matter to divide pixels dimensions by the SVGAtlas.SPRITE_PIXELS_PER_UNIT

constant value. To perform the needed conversions, we have exposed some useful properties in the SVGCameraBehaviour script:

```
public float WorldWidth {
    // get the camera viewport width, in world coordinates
    get {
        return (this.camera.pixelWidth / SVGAtlas.SPRITE_PIXELS_PER_UNIT);
    }
}
public float WorldHeight {
    // get the camera viewport height, in world coordinates
    get {
        return (this.camera.pixelHeight / SVGAtlas.SPRITE_PIXELS_PER_UNIT);
    }
}
```

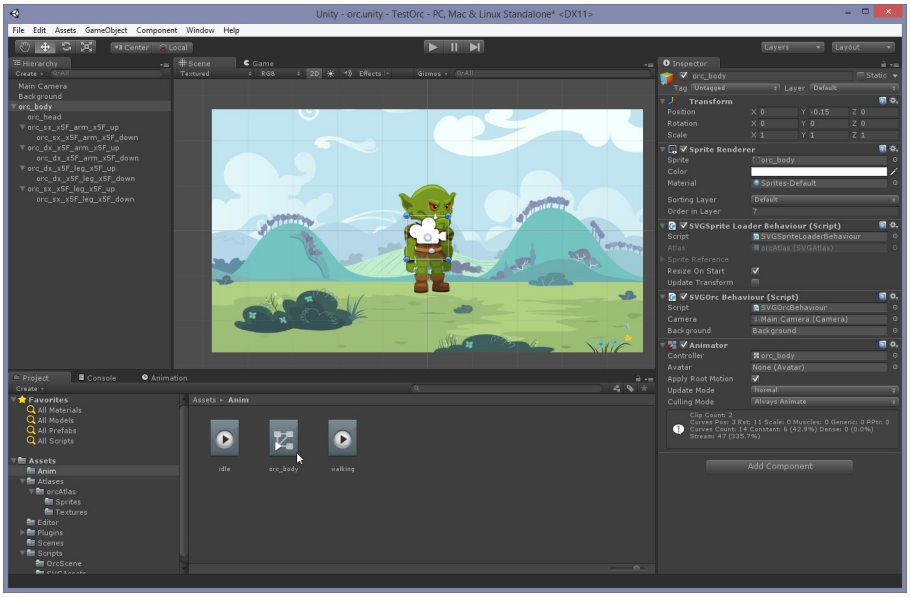
and in the SVGBackgroundBehaviour script:

```
public float WorldWidth {
    // get the background sprite width, in world coordinates
    get {
        return (this.PixelWidth / SVGAtlas.SPRITE_PIXELS_PER_UNIT);
    }
}
public float WorldHeight {
    // get the background sprite height, in world coordinates
    get {
        return (this.PixelHeight / SVGAtlas.SPRITE_PIXELS_PER_UNIT);
    }
}
public float WorldLeft {
    // get the background sprite x-coordinate relative to its left edge, in world coordinates
    get {
        return this.gameObject.transform.position.x - (this.WorldWidth / 2);
    }
}
public float WorldRight {
    // get the background sprite x-coordinate relative to its right edge, in world coordinates
    get {
```

```
        return this.gameObject.transform.position.x + (this.WorldWidth / 2);
    }
}
```

In order to complete the example, we must animate the orc; we can use the “idle” and “walking” animations that we have already prepared in the “Assets/Anim” folder:

- select the orc_body object
- add an Animator component (menù Component → Miscellaneous → Animator)
- drag&drop the “orc_body” controller from the Assets/Anim to the Animator's Controller field



Now click Play and see the animated orc, starting with the “idle” animation; by pressing the mouse (or tapping the device screen), the orc will walk towards right or left.

6. Requirements and limitations

SVGAssets supports a subset of SVG Tiny 1.2 relative to static vector graphics rendering, limited to Structure Module, Style Module, Shape Module, Gradient Module, plus some SVG Full 1.1 features:

- **<svg> supported with the exception of <svg> element nesting**
- **<g> supported (Tiny 1.2, no group opacity)**
- **<defs> supported**
- <desc> unsupported (simply skipped)
- <title> unsupported (simply skipped)
- <metadata> unsupported (simply skipped)
- **<use> supported (Tiny 1.2)**
- <symbol> unsupported (simply skipped)
- **<style> supported (Tiny 1.2 + inline styles)**
- **<circle> supported**
- **<ellipse> supported**
- **<line> supported**
- **<path> supported (Tiny 1.2 + elliptical arcs)**
- **<polygon> supported**
- **<polyline> supported**
- **<rect> supported**
- **<linearGradient> supported (Tiny 1.2 + spread modes)**
- **<radialGradient> supported (Full 1.1)**
- **<stop> supported**

SVGAssets plugin supports rendering surfaces with dimensions less than or equal to 4096 pixels only.

7. Credits

Credits for SVG arts included in SVGAssets package go to:

- "Ror362", for its great girl.svg

  Website: http://ror362.deviantart.com/art/Yayoi-iM-S-397886689

- Casper Tang Veje ("adcoon"), for its beautiful background.svg

  Website: http://adcoon.deviantart.com

- Chris Hildenbrand, for its awesome orc.svg

  Website: http://2dgameartforprogrammers.blogspot.com

Sunny countryside backgrounds (gameBkg1.svg, gameBkg2.svg, gameBkg3.svg, gameBkg4.svg) have been downloaded from:

- http://xoo.me/it/template/details/6960-4-sunny-countryside-vector-scenes