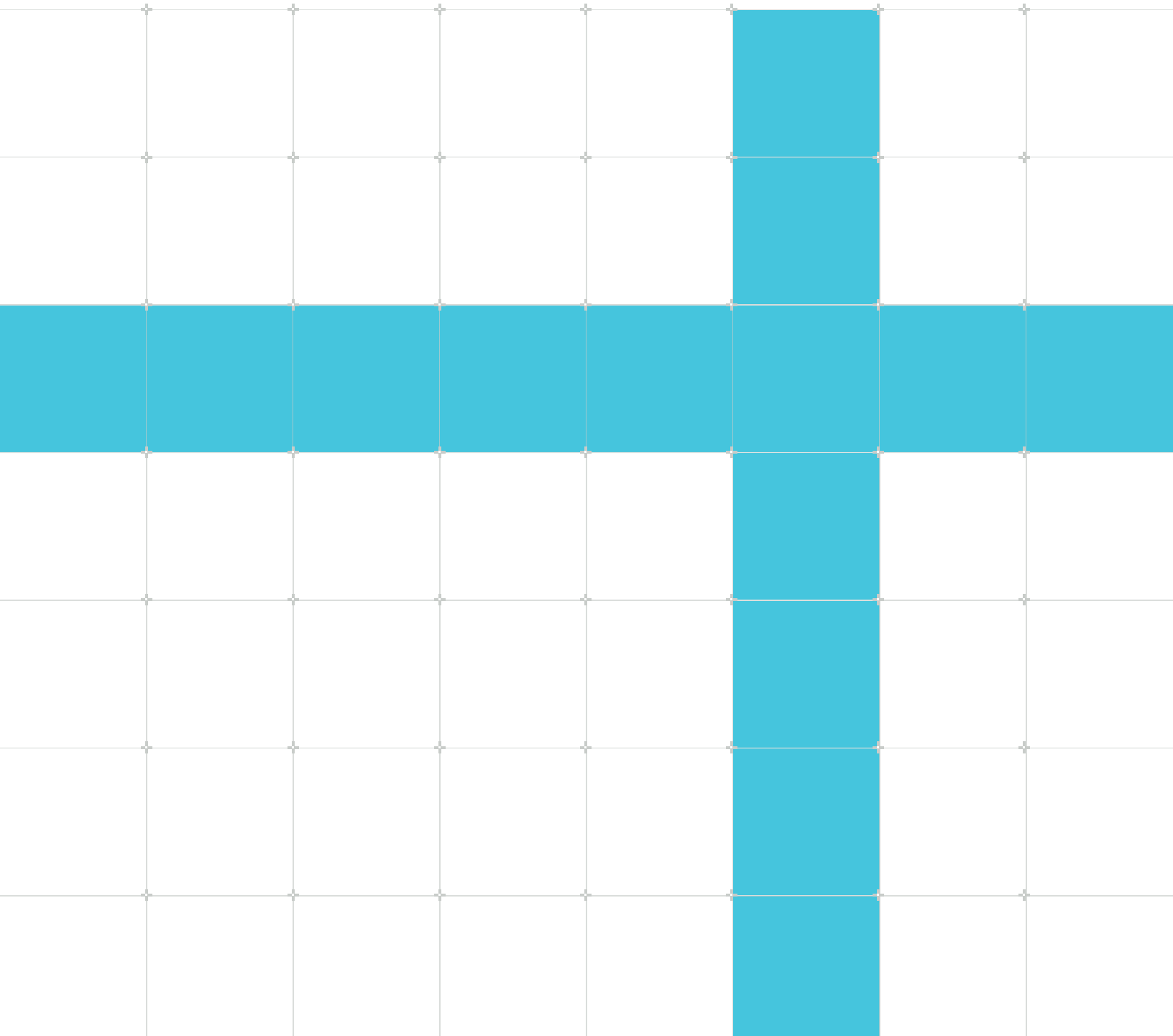
Real-time 3D Art Best Practices - Geometry

Version 1.0

Issue 02
102448_0100_02_en



Real-time 3D Art Best Practices - Geometry

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	31 March 2020	Non-Confidential	Initial release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

To make sure that a game runs well on all devices, the geometry considerations of a game must be taken seriously and optimized as much as possible.

This guide highlights geometry optimizations for 3D assets that can make a game more efficient. These optimizations help achieve the overall goal of improving your performance of your game on mobile platforms.

This guide is also available in the format of a Unity Learn Course - Arm & Unity Presents: 3D Art Optimization for Mobile Applications.

At the end of this guide, you can Check your knowledge. You will learn:

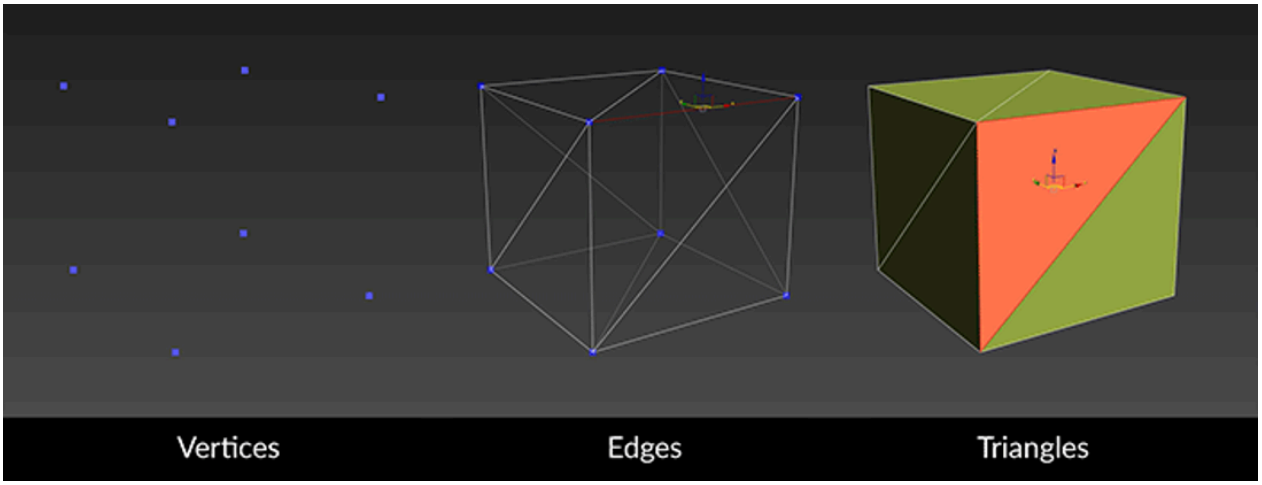
- The fundamental components of geometry
- How geometry contributes to the performance of your game
- How to use different techniques like Level of Detail to optimize geometry

2. What is geometry?

Geometry, or a polygon mesh, is a collection of vertices, edges, and faces that make up the shape of a 3D object. This 3D object can be a car, weapon, environment, character, or any type of asset that appears in a video game.

The following diagram shows the three elements that make up the geometry of a 3D object:

Figure 2-1: Elements of a 3D object



The elements are as follows:

- Vertices are the points that make up the surface of a 3D object.
- Edges connect two vertices with a straight line.
- Triangles consist of three vertices that are connected to each other by three edges. Other terms for a triangle include polygon and face.

Note

With 3D software like Max, Maya, or Blender, you often use a four-sided polygon called a quad. Quads can be easier to modify and work within these 3D programs. When rendered on-screen, all of these polygons are displayed as triangles.

3. Triangle and polygon usage

To optimize the performance of your game, you must be aware of the number of triangles that are on-screen at any given time.

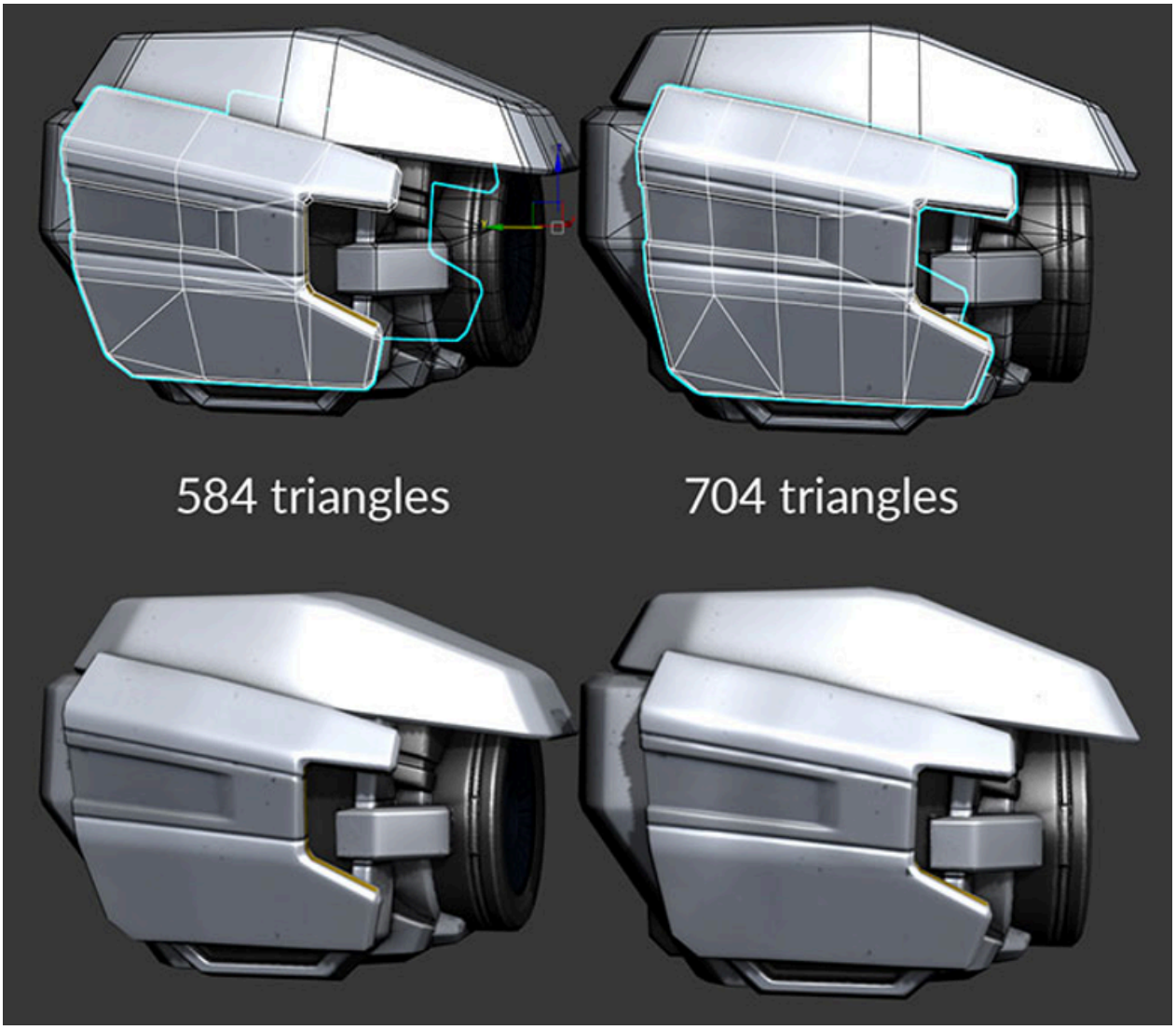
Use the minimum number of triangles to get the correct balance between the intended quality of your 3D objects, or models, and delivering consistent performance.

We recommend that you try the following tips:

- Using fewer triangles results in increased performance.
  - It is essential to consider the triangle count when creating content for a mobile platform.
  - Fewer triangles mean fewer vertices for the GPU to process.
  - Processing vertices is computationally expensive. Fewer vertices to process results in better overall performance.
- Using fewer triangles enables the release of the game on more devices, and not just the latest devices with the most powerful GPUs.

The following image shows a comparison between two 3D objects. One object uses 584 triangles, and the other object uses 704 triangles. Both objects look the same in shaded mode. This shows that you can remove any edges in your models that do not contribute to the silhouette.

Figure 3-1: 3D object comparison



In Unity, the format of the mesh index buffer dictates the maximum number of vertices that each 3D object can use:

- A 16-bit index buffer supports up to 65,535 vertices
- A 32-bit index buffer supports up to 4 billion vertices

Some older GPUs, including Android devices using the Mali-400, only support 16-bit index buffers. These devices do not render 3D objects with more than 65,535 vertices. For maximum cross-platform compatibility, use 16-bit index buffers and keep the number of vertices in a single mesh under the limit.

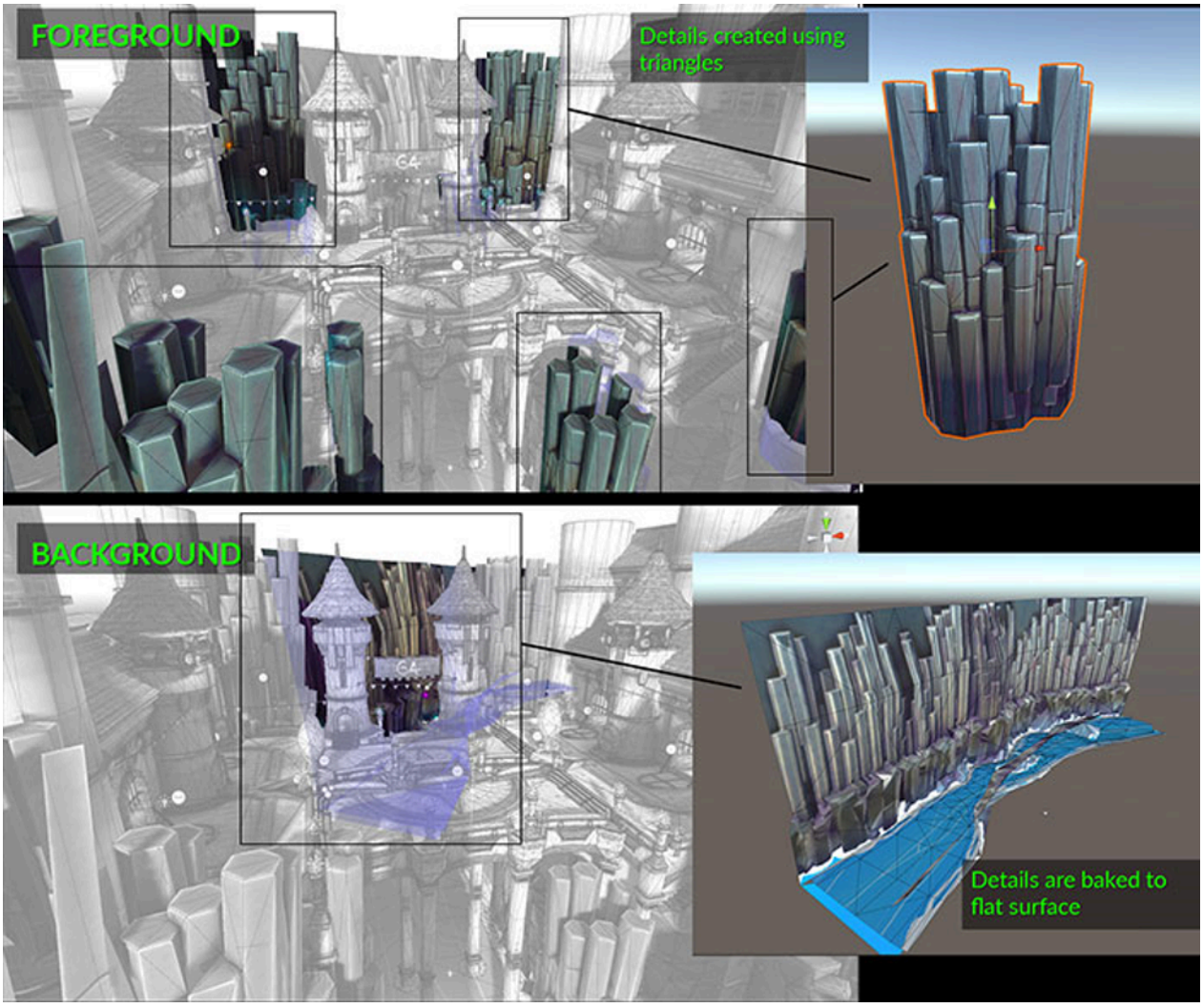
It is essential to view or test the game on as many target devices as possible. Testing your game on just a computer screen does not give you the information that you need for optimization.

Keep in mind that mobile device screens are smaller than the average computer monitor. Therefore, any details that are created using lots of triangles may not even be visible on a mobile device.

Use more triangles on 3D objects that are in the foreground and therefore closer to the camera. Use fewer triangles on 3D objects that are further away in the background. A game that uses a static camera Point-of-View (POV) benefits more from this technique.

The following image shows an example where 3D models are used in the foreground, but lower quality 3D models are baked into the 2D background:

Figure 3-2: 3D models in foreground



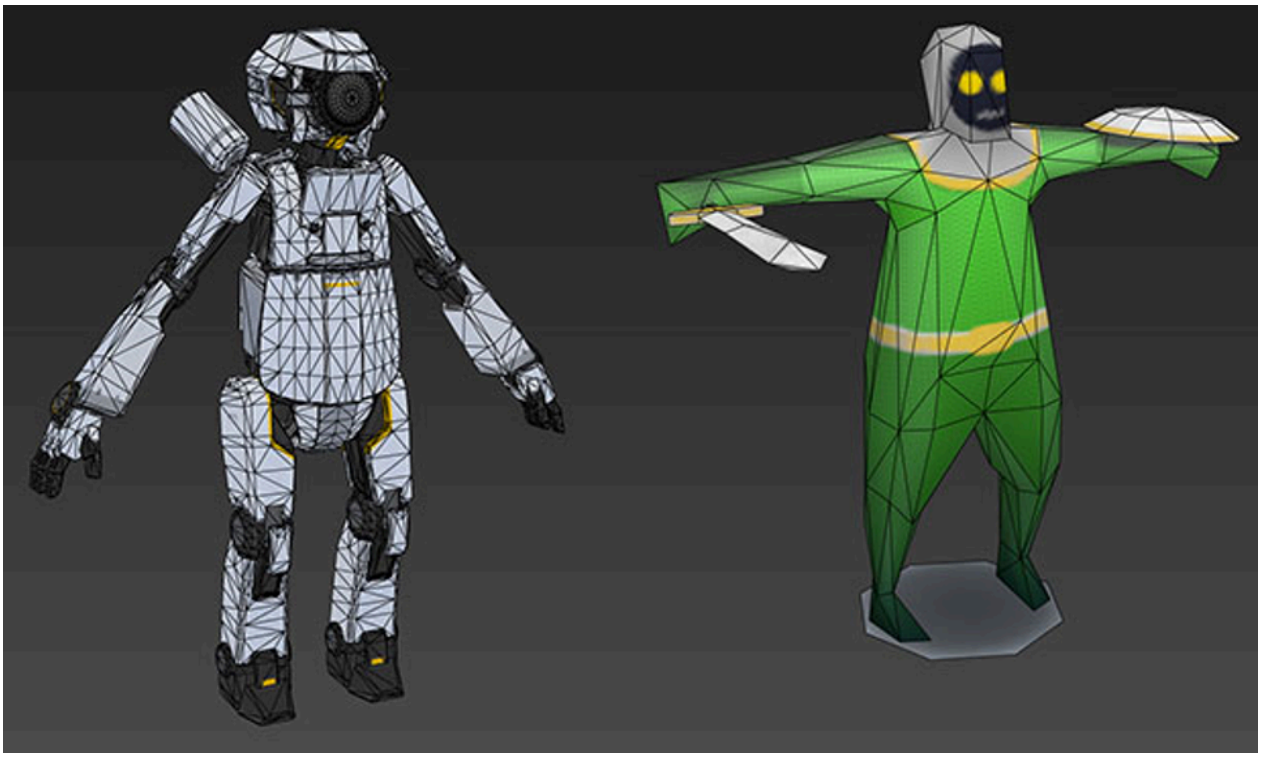
While there are no fixed numbers for the maximum on-screen triangle count, the more 3D objects that are on-screen at the same time, the fewer triangles you can use per object. However, if you are displaying fewer 3D objects on-screen, then you can use more triangles per object.

The target device also matters. Newer phones are able to handle more complex geometry than older mobile devices.

The following example image shows characters from two demos:

- The Circuit VR demo only has one robot character, so a model with a higher polygon count can be used as shown on the left.
- The Armies demo has hundreds of soldiers in one frame, so it needs a much smaller per-object polygon count, as shown on the right.

Figure 3-3: Character demos



The Armies demo is a 64-bit mobile device demo that was built in Unity. In this demo, the camera is static with lots of animated characters. Each frame renders approximately 210,000 triangles in total. This triangle count enables the demo to run smoothly at approximately 30 Frames Per Second (FPS).


Note

The number of triangles to use depends on both the type of game that is being created, and the specifications of the target devices.

The following example shows the total number of triangles that were used in the Armies demo:

Figure 3-4: Number of triangles in Armies demo

The largest objects in the scene, the cannon towers, are approximately 3000 triangles each. This is because they occupy a large portion of the screen.

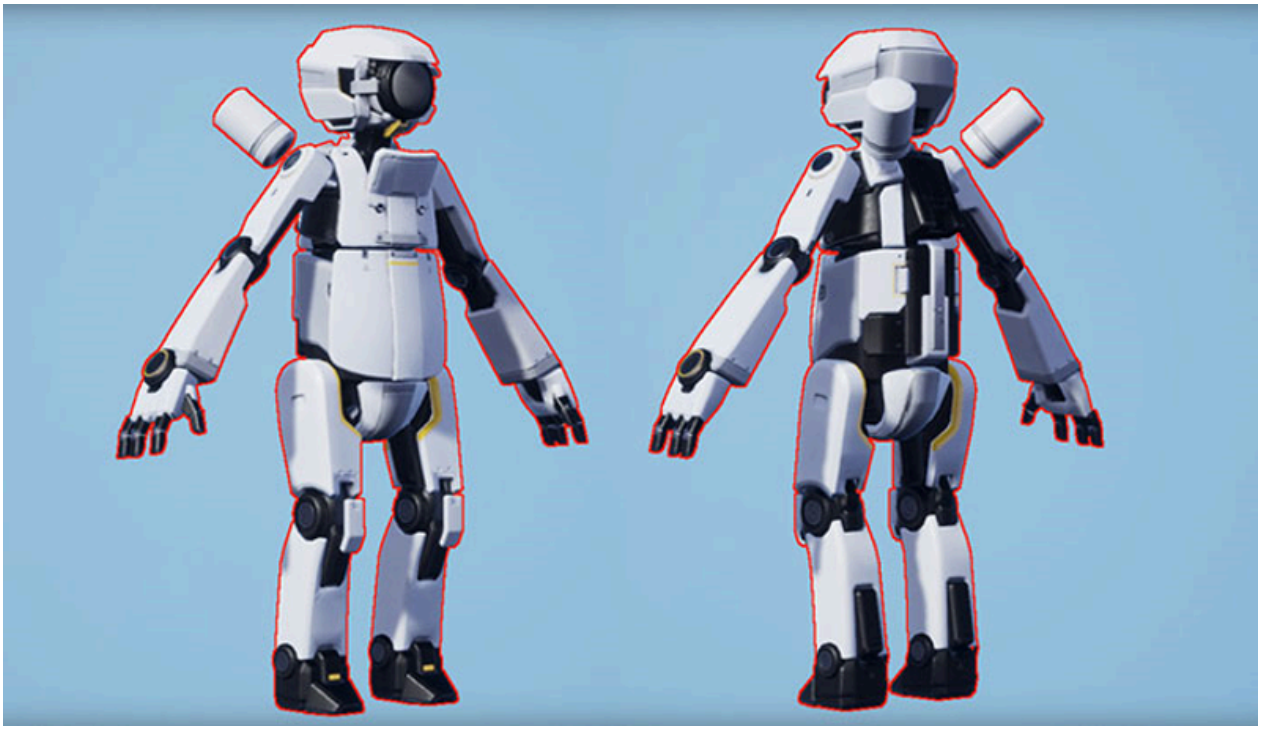
Characters only use approximately 360 triangles. This is because there are many characters, and they are only seen from a distance. From the camera point-of-view, the characters look fine on screen.

Triangle distribution triangles in areas that matter

Both polygons and vertices are computationally expensive on mobile platforms. By placing polygons in areas that really contribute to the visual quality of the game, we are not wasting the processing budget.

Due to the small screen size on most devices and the location of 3D objects in your game, many small triangle details on a 3D object might not be visible in-game. This means that you should focus on large shapes and parts that contribute to the silhouette of the object, rather than small details that might not be visible. The following image highlights the object silhouette in red, to show how the different shapes contribute to it.

Figure 3-5: Object silhouette



Use fewer triangles on the areas that are not often shown on-screen. Examples include the bottom of a car, or the back of a wardrobe.

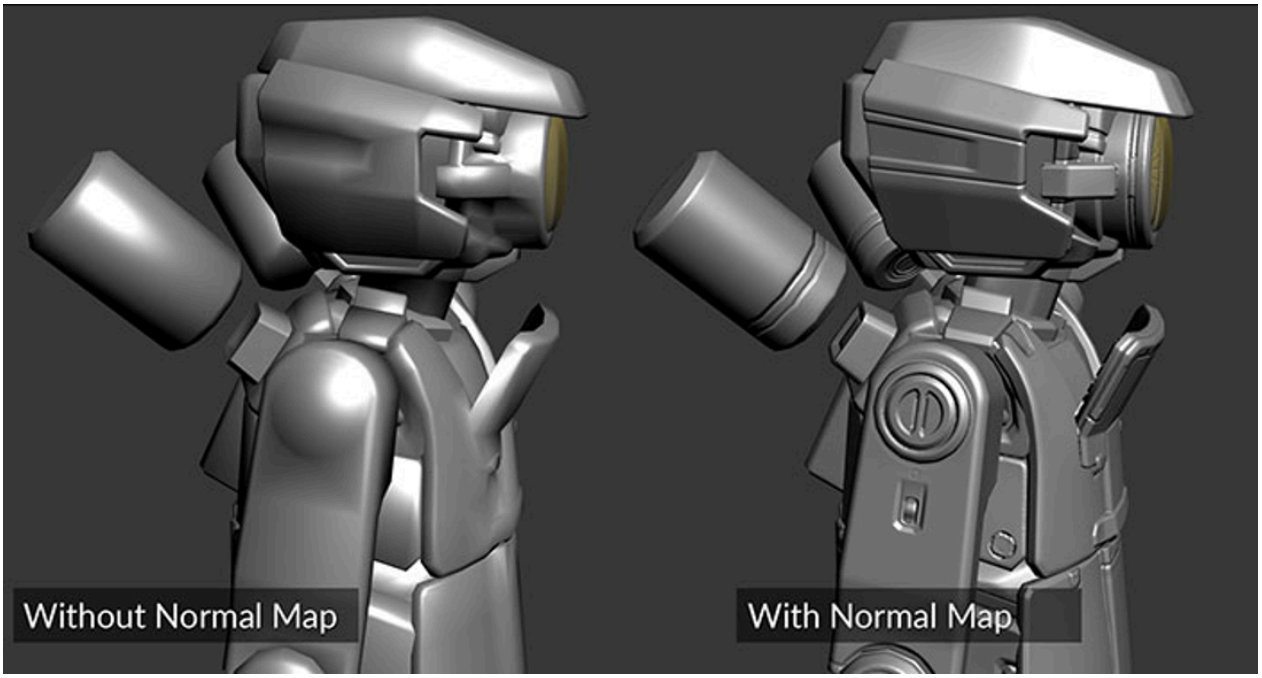
Avoid using high-density triangle meshes for modeling small details. Instead, use textures and normal maps for fine detail.

Note

A normal map is a texture map that stores the surface direction at each pixel.

The following images show the same mesh with and without normal map:

Figure 3-6: Mesh with and without normal map



Consider deleting the back, or bottom, part of an object that is never seen from the camera POV.

However, deleting object parts needs to be done carefully, because it might limit the re-usability of the scene. For example, you might delete the bottom part of a table mesh, because you think that it is never seen by the end user. Doing this would mean that you could no longer place that model upside down, or reuse that model in another game where the underside of the object would be seen.

Why you should avoid micro triangles

Micro triangles are tiny triangles that do not contribute much to the final look of an object or scene.

When a 3D object with a large polygon count is moved further away from the camera, a micro triangles problem occurs. Micro triangles are triangles on a device that are between one and ten pixels in size.

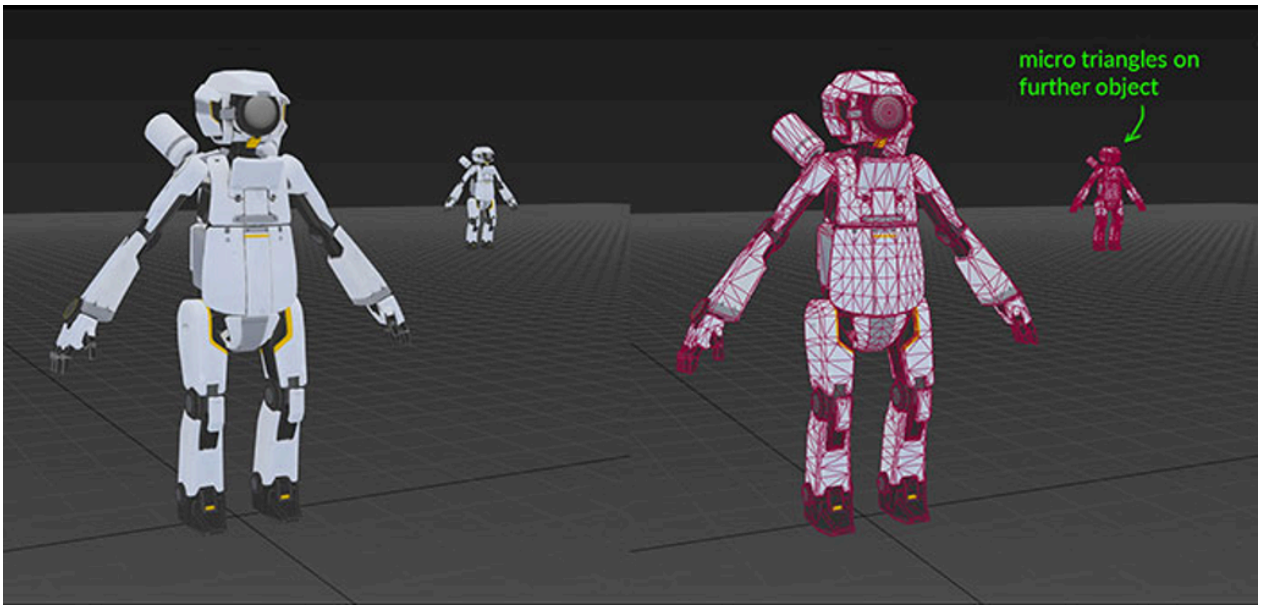
Micro triangles reduce performance because the GPU must process all of these triangles, even though they are too small to see in the final image. Remember that vertices are computationally expensive to process, and more triangles on-screen at the same time means more vertices to process.

The following two approaches cause micro triangles:

- Details that are too small and consist of many triangles
- Objects with many triangles that are further from the camera

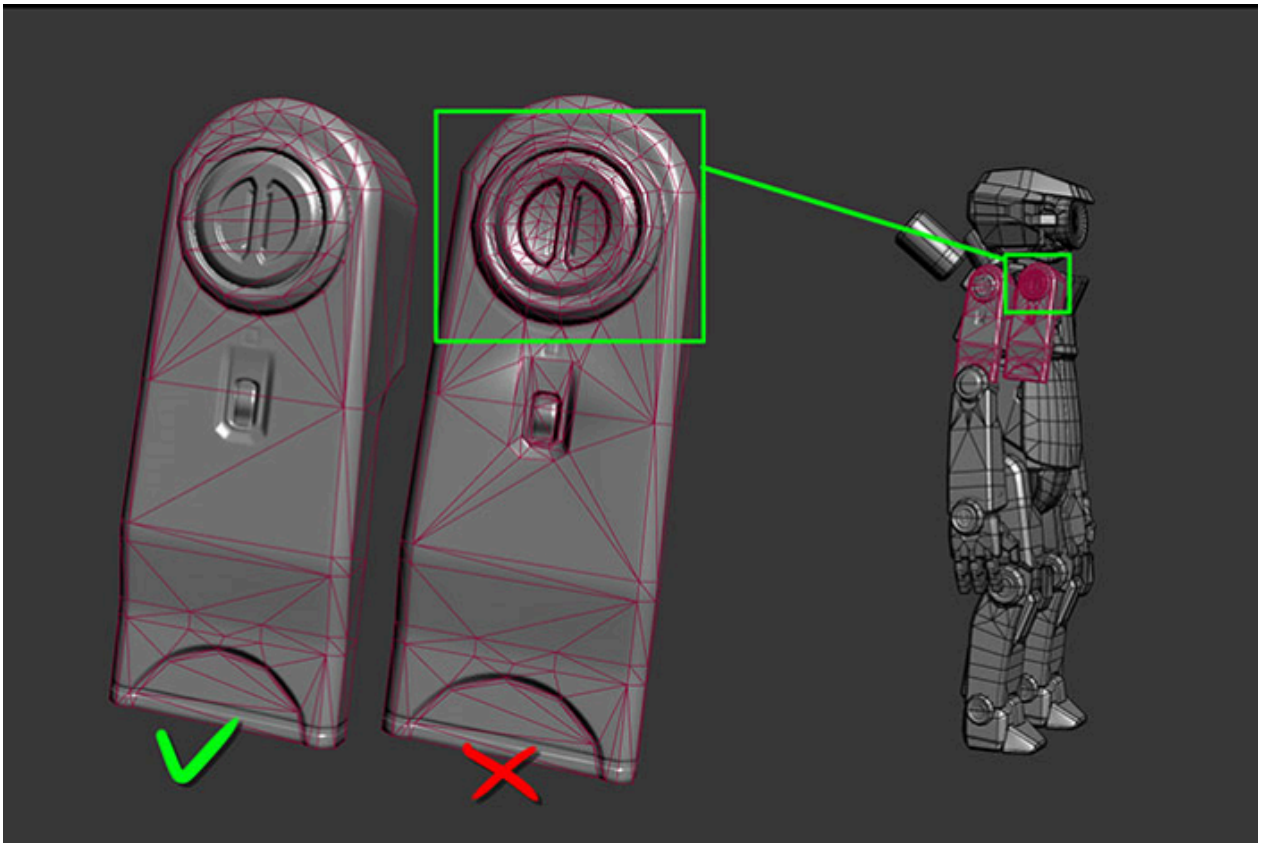
The following image shows the number of triangles that are used when a 3D object is both nearer to, and further from, the camera. The gray images on the left use fewer triangles. The gray and red images on the right uses a normal map to display the same amount of visible detail:

Figure 3-7: Micro triangles on further objects



In the following image, most of the triangles in the highlighted area are too small to be visible on a mobile device. Therefore, they do not contribute much to the final look of the image:

Figure 3-8: Micro triangles close-up



How to minimize the micro triangle problem

Here are a few steps that you can take to mitigate problems with micro triangles:

- For an object that changes its distance from the camera, use Level of Detail (LOD). Using the correct LOD simplifies an object when it is further away, and the object uses fewer triangles.
- Use fewer triangles on background objects.
- Avoid using polygons to create the finer details in a mode. Instead, use a combination of textures and a normal map instead.
- Merge any vertices or triangles that are either too small to see on-screen, or are not adding much value to the final image.
- Try to keep triangles above ten pixels in area.

Minimizing the use of micro triangles

There are several reasons to minimize the use of micro triangles:

- The GPU must process all triangles and vertices, even when they do not add any value to the final scene on-screen.
- Memory bandwidth is negatively affected, because more data must be sent to the GPU for processing.

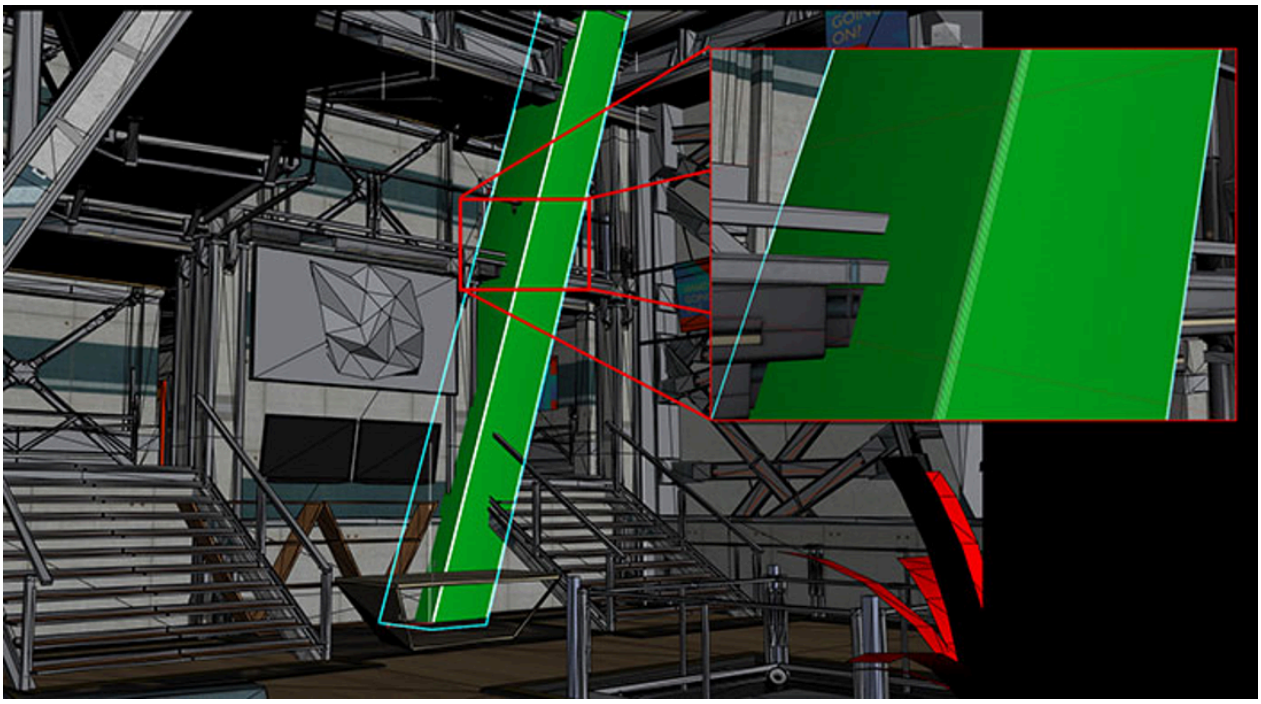
- The amount of processing that is required directly impacts the battery life of a mobile device. Therefore, the more data that the GPU must process, the less time the battery lasts.

Why long and thin triangles can cause problems

Long, thin triangles are vertices that, when rendered in the final image, are smaller than ten pixels. Long, thin triangles can cause issues because they are more expensive for the GPU to process when compared with normal triangles. This is because GPUs process pixels in square blocks. Long, thin triangles can mean that the GPU has to perform calculations for many square blocks, even though the actual number of pixels that the thin triangle occupies is relatively small. This situation results in wasted computations.

The following screenshot shows an example of a long, thin triangle in use. The screenshot highlights the bevel on the pillar when it is viewed from a distance. However, the bevels are not a problem when seen close up:

Figure 3-9: Long and thin triangle



Minimize the long thin triangle problem

Here are a few steps that you can take to mitigate issues with long, thin triangles:

- Remove any long, thin triangles that you see from all objects when possible. The best solution is to remove the long, thin triangles completely, although there can be situations where this is not possible.
- Avoid using shiny materials on an object with long, thin triangles, because this causes flickering.
- Use Level of Detail (LOD) and remove the long, thin triangles when they are further away on-screen.

- Try and keep triangles close to equilateral. Ensure that the triangles have more inside area when compared to the edges.
- A more detailed explanation of this problem can be found here.

4. Level of Detail

Level of Detail (LOD) is a technique to reduce mesh complexity as objects become more distant from the viewer.

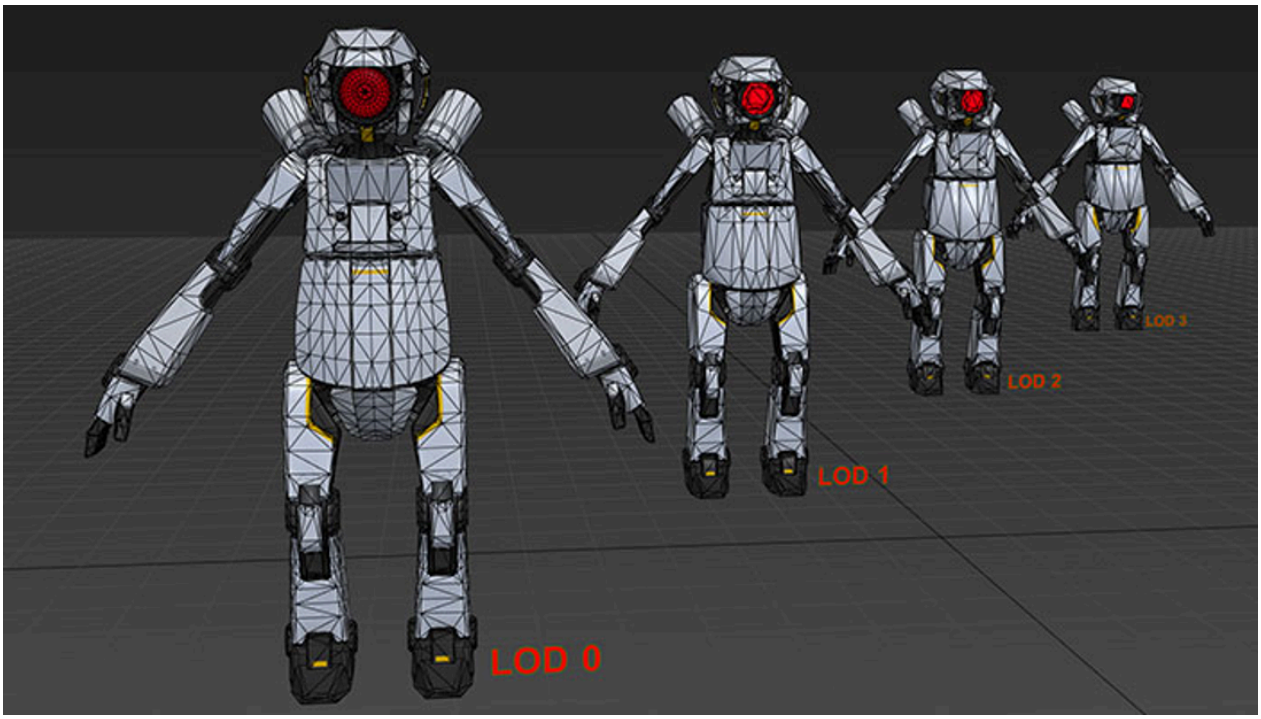
LOD provides the following benefits:

- LOD reduces the number of vertices that must be processed.
- LOD avoids the micro triangles problem.
- LOD often looks better for objects that are placed further away in the scene.

Arm recommends that you optimize the LOD for every 3D object that has significant changes in distance from the camera.

The following image shows how you can use LOD management to reduce the complexity of a 3D model, while retaining an appropriate level of detail as the model moves further away from the camera:

Figure 4-1: Level of detail management



Here are a few other things to remember when considering how to optimize your use of LOD:

- Consider how triangles affect the silhouette of objects when you are reducing triangle counts for LOD.
- LOD can also apply to shader complexity. The shader and material can be optimized for 3D objects that are further away. For example, by reducing the number of textures that are used.

- Remove more polygons on flatter areas.
- Use mipmaps as LOD for textures.

When to avoid LOD

LOD is not suitable in every situation. For example, avoid using LOD in a game where both the camera view and objects are static, or where the object is already using a low polygon count.

LOD comes with a memory overhead, and a larger file size. All of the LOD mesh data must be saved in memory, so that this data can be utilized in real time.

The following image shows a scene where LOD is not used because the scene is static. Instead, you can use other optimization tricks, like removing polygons that are never visible to the player:

Figure 4-2: When to avoid LOD

Why use LOD

As an object goes further away from the camera, you can see less of the detail of that object. From twenty meters away, it is hard to see any difference between one version of an object with 200 triangles, and another version of the object with 2,000 triangles. Therefore, it is not necessary to use extra triangles that add nothing of value to the scene.

A few other key benefits of using LOD include:

- A boost in performance because fewer triangles must be processed
- Mitigation of problems that micro triangles cause

The following image shows you how distant objects look the same, even with different polygon counts:

Figure 4-3: Why use LOD

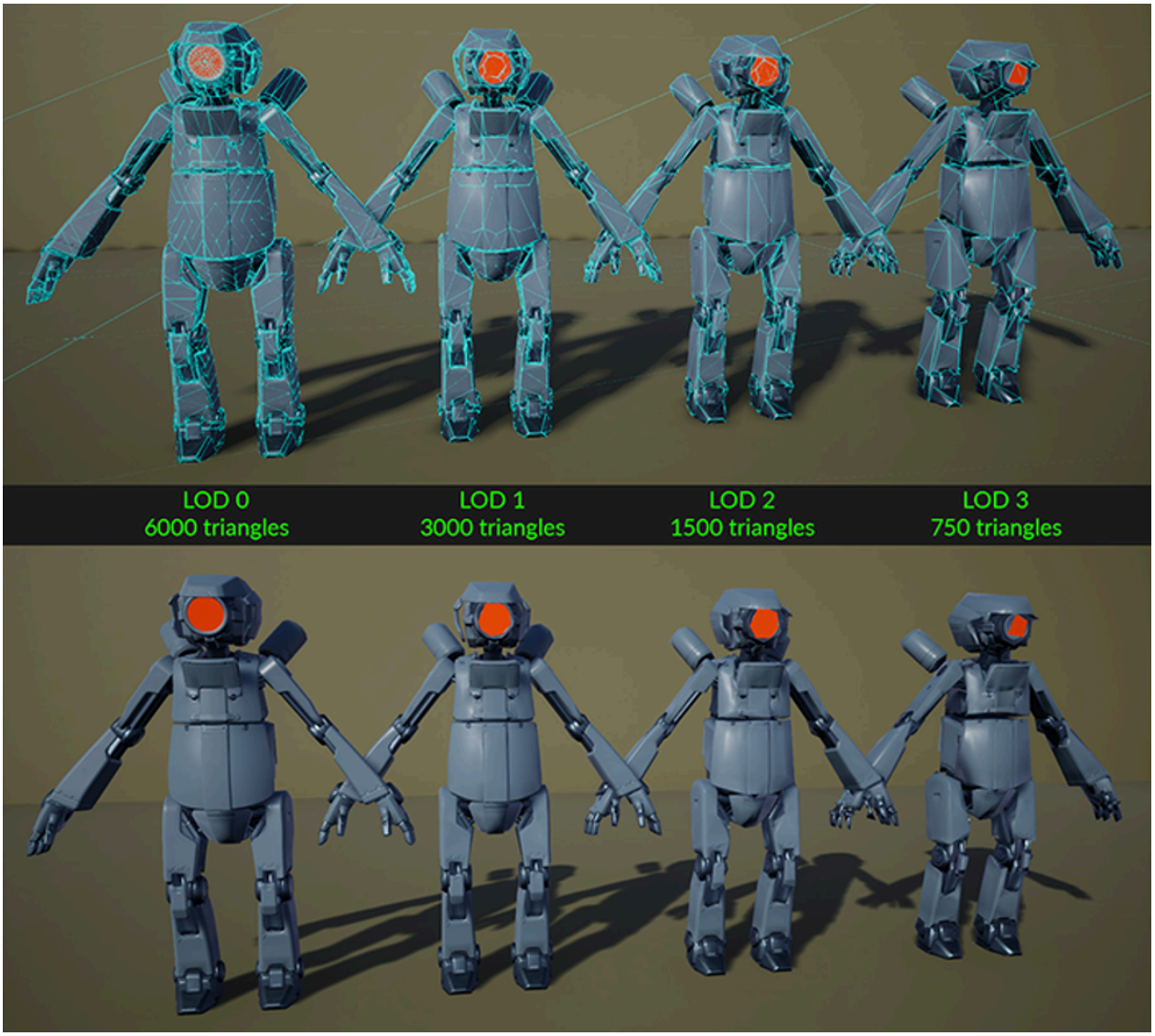
Use an appropriate number of triangles

Here are some key points to consider when you decide the number of triangles in each LOD:

- It is often worth reducing the number of triangles between each LOD level by 50%.
- Do not use a dense number of triangles on lower LODs. These lower LODs are only seen when the object is further away.
- Check what the LODs look like at the correct distances from the camera on which they are meant to be seen. Lower LODs have lower resolution when viewed up close. This is okay, because they are not meant to be seen up close.

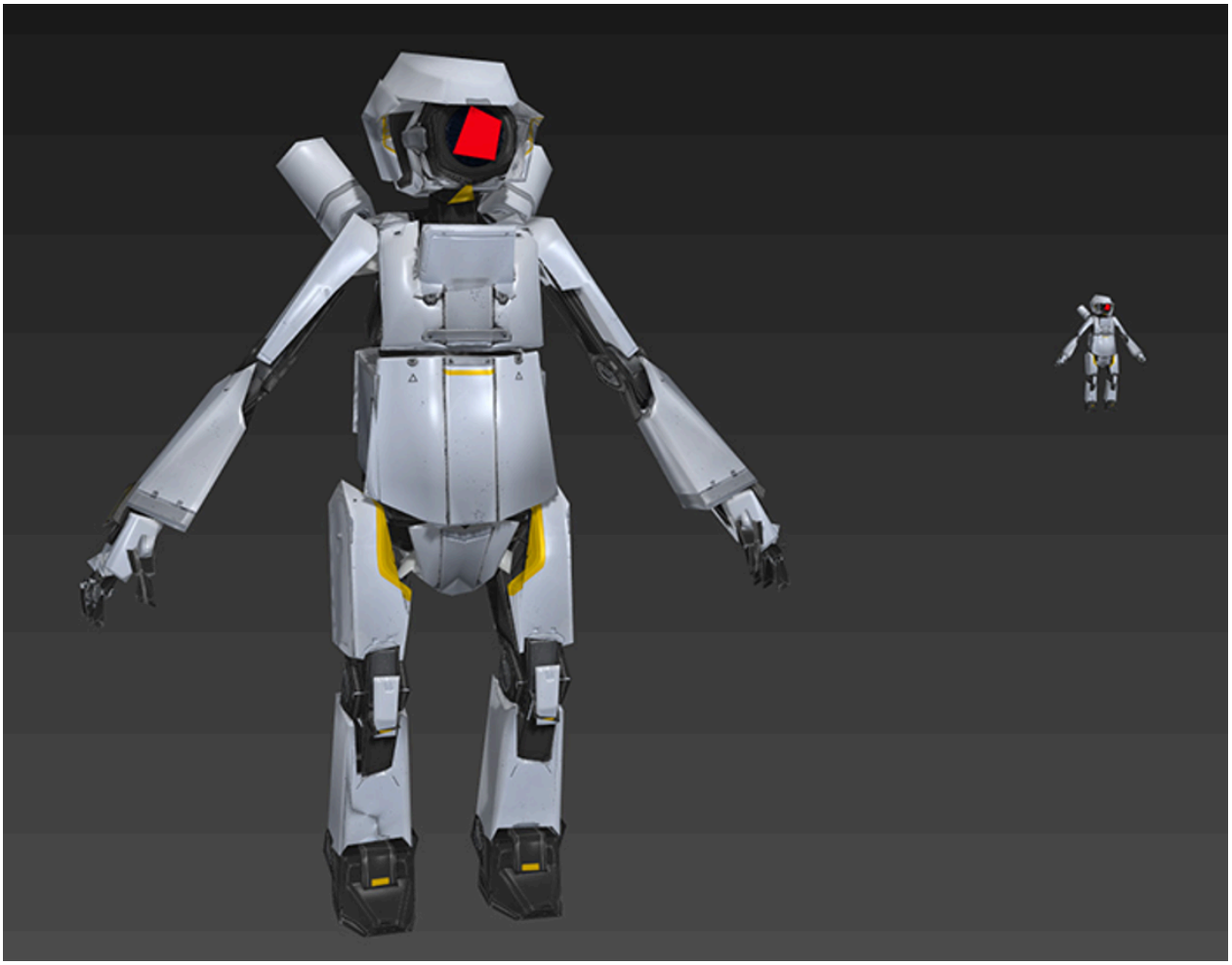
The following image shows you how an object can look when the number of polygons that are used drops by 50% for each level:

Figure 4-4: Number of triangles



In the following image, the lower LOD object looks poor when viewed up close. However, their appearance is acceptable when they are viewed at the correct distance from the camera:

Figure 4-5: Low LOD object



LOD triangle counts matter. This is because, if you do not reduce the polygon count enough on lower LOD objects, the performance of the game is negatively impacted. This is because the CPU is processing more vertices than is needed.

However, if you reduce the polygon count of your lower LOD objects too much, items look like they are popping in and out of detail in real time. This popping effect is noticeable to users, and could ruin their immersion in your game.

What is a reasonable number of LOD levels to use?

There is no precise number for how many LODs an object can have. It depends on both the size and importance of the object. For example, a character in an action game, or a car in racing game, can benefit from using more LOD levels than a small background object, like a tree.

If too few LOD levels are used, the effect would be:

- Reduced performance gain if the polygon reduction is not substantial enough between LOD levels.

- If there is too large a jump in polygon reduction, then the popping that happens on LOD switch is more noticeable.

If too many LOD levels are used, the effect is:

- Increased workload for the CPU, because more processing is needed to decide which LOD to display.
- Increased memory usage and larger file sizes to store the extra meshes.
- More time is required to create and verify the LOD models, especially if an artist creates them by hand. This is the largest cost.

Creating LOD meshes

hen creating lower LOD meshes manually in 3D software from a higher LOD mesh, remove edge loops or reduce the number of vertices on a 3D object. While this gives more control to the artist, it can take a longer time to do.

When creating LOD meshes automatically, use a built-in modifier, or separate LOD generation software. As an example of built-in modifiers, in 3DS Max, use the ProOptimizer function, or the Generate LOD Meshes function within Maya.

Implementing LOD in Unity

For guidance on how to implement LOD in Unity, read Implementing Level of Detail in Unity.

5. More geometry best practices

There are a few more techniques and tricks that you can try to help optimize the performance of your game even further.

Smoothing groups

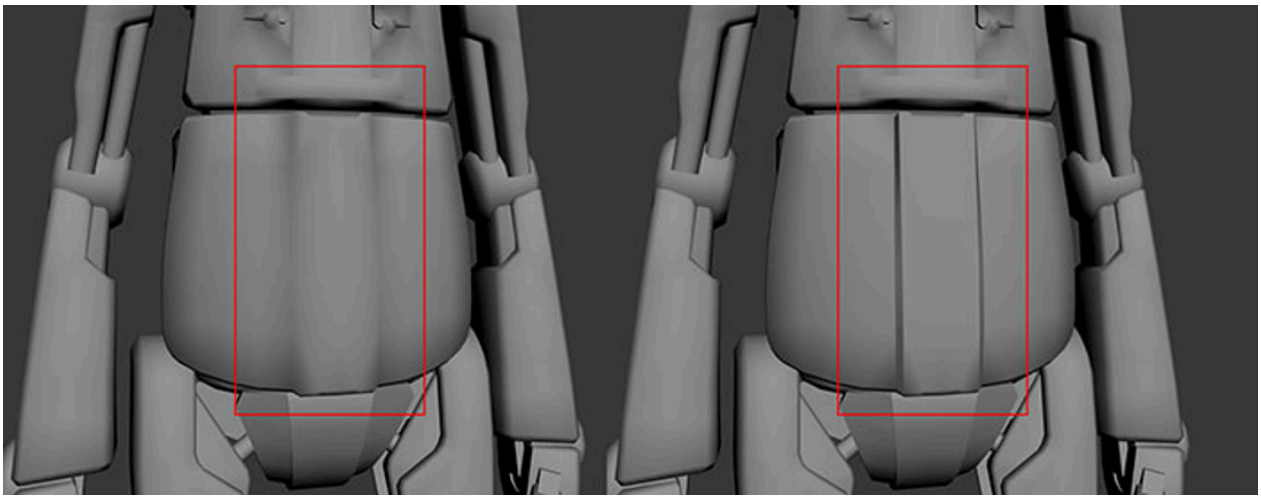
Use smoothing groups, or custom vertex normals, to define the hardness of an edge and alter the look of a model. Smoothing groups helps to create better shading when the art direction intentionally uses a tiny polygon count.

Take extra care, because smoothing groups affect the UV islands of a model, and also the quality of a normal map when you do baking. For more information, see the Real time 3D Art Best Practices Texturing.

If smoothing is implemented on a 3D model, the model must be exported from the 3D software and imported into the engine.

The following image shows an example of how smoothing works when applied to an object. The same model is shown before smoothing on the right, and after smoothing on the left:

Figure 5-1: Smoothing groups



Mesh topology

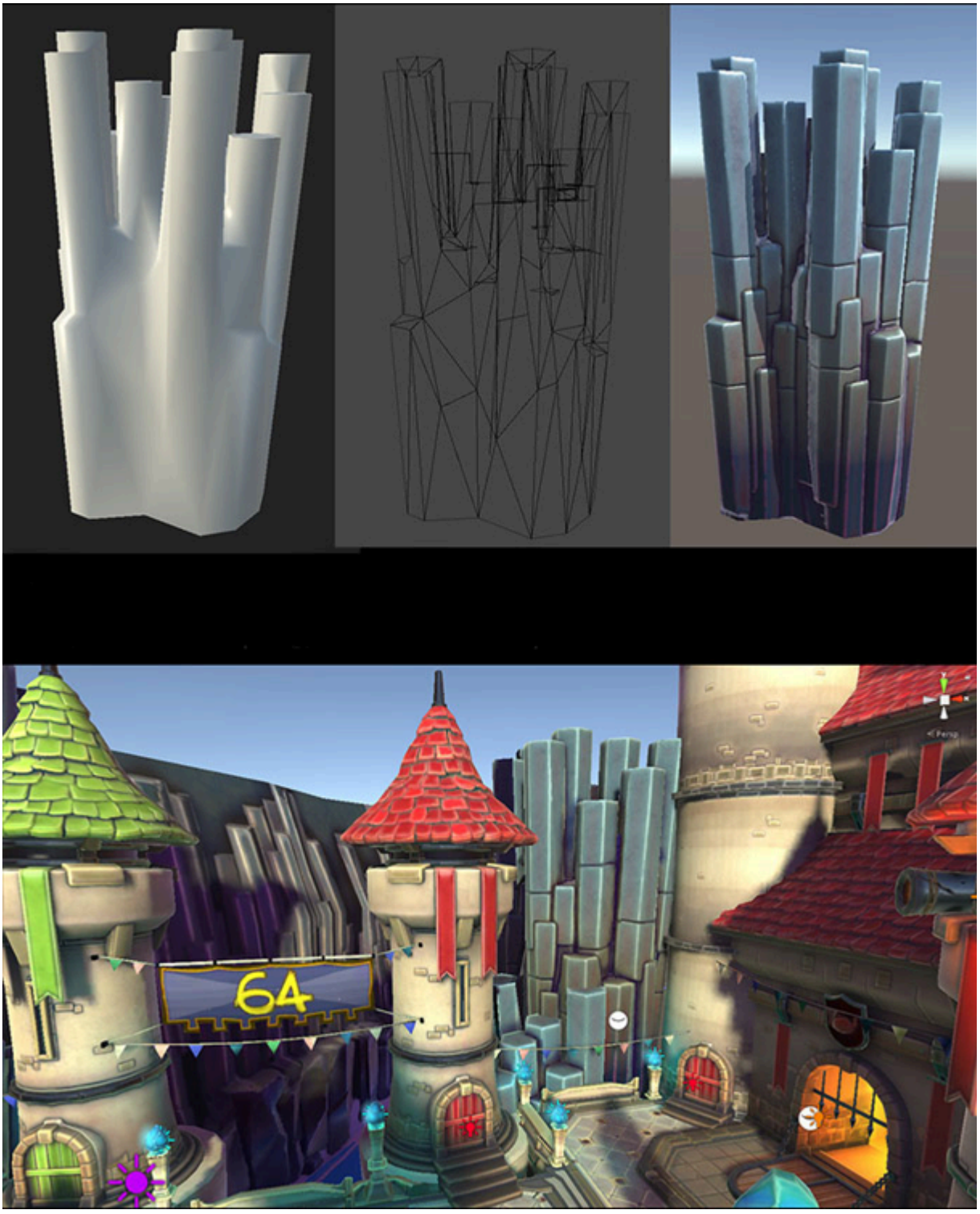
Remember these tips when using mesh topology:

- When creating a 3D asset, use a reasonably tidy and clean topology.
- Clean topology is essential for characters, or other objects, that are deforming or animated.
- Do not be obsessed with having perfect topology on a 3D model. While not all objects need a perfect edge flow, try to keep the model tidy. Keep the following points in mind:
    - The player, or end user, does not see the wireframe of a 3D model.

- The texture and material that is applied to the mesh have a bigger contribution to the look of a 3D model.

The following image shows an example of a wireframe of the rock cliff mesh that uses simple geometry and topology. The rock cliff looks much better with the material that has been applied. Therefore, any issues with the topology have disappeared:

Figure 5-2: Mesh topology



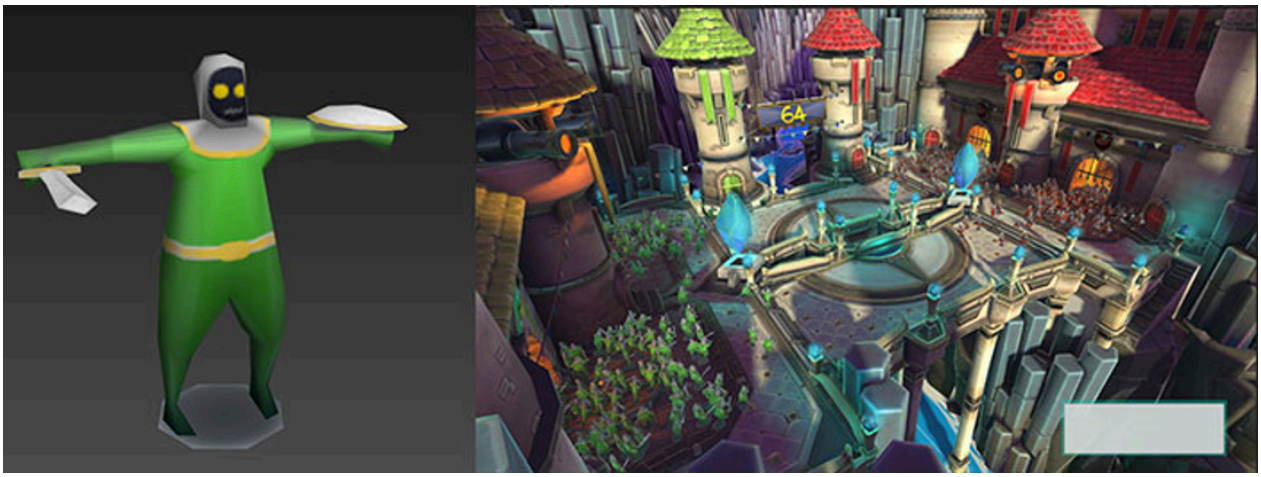
Shape exaggeration

Shape exaggeration is a technique where certain parts and shapes are made bigger than normal to help with readability. When to use shape exaggeration depends on the type and style of the game that you are creating.

For example, a mobile device screen is small and it is sometimes hard to capture certain shapes when they are tiny. Exaggerating the shape helps to overcome this. If you make the hands on a character larger so that they are easier to see on a small screen, you will improve the experience of your users.

The following image shows how an exaggeration in the size of a hand could look. The hand, sword, and general body proportions are all emphasized in different ways, as you can see in the left side of the image. This was done to improve the visibility, while accounting for the lower polygon count that was used. The right side of the image shows how the final scene looks with the exaggerated character shapes:

Figure 5-3: Shape exaggeration



6. Check your knowledge

The following questions will help you test you knowledge:

What are the three components of object geometry?

Vertices, edges, and triangles.

For best compatibility with older devices, what is the maximum number of vertices that a 3D object should use?

Older GPUs, including Android devices using the Mali-400, only support 16-bit index buffers which provide up to 65,535 vertices.

Using Level of Detail (LOD) can improve performance by reducing mesh complexity when objects are more distant from the camera. What are the disadvantages of using LOD?

- Increased CPU workload to calculate which LOD to use
- Increased memory and file size for the extra models
- Increased cost to create the different models

7. Related information

Here are some resources related to material in this guide:

- Arm Developer: Graphics and Gaming Development
- Implementing Level of Detail in Unity
- Humus blog post: Triangulation

8. Next steps

This guide introduces geometry optimizations for 3D assets that can make a game more efficient. These optimizations help to achieve the overall goal of improving the performance of your game on mobile platforms.

You can continue learning about best practices for game artists and how to get the best out of your game on mobile by reading the other guides in the series:

- Real-time 3D Art Best Practices: Materials and Shaders
- Real-time 3D Art Best Practices: Textures