



Implementing culling in Mali GPU User Interface Engine applications

Document Number: PR398-PRDC-011905 1.0
Date of Issue: 28th June 2010
Product: Mali GPU User Interface Engine v2.3.1

© Copyright ARM Limited 2010. All rights reserved.

Abstract

Release 2.3.1 of the Mali GPU User Interface Engine (the UI Engine) includes a new function for getting the bounding box for an object in a scene. This paper discusses how this new function can be used to implement culling, and hence to improve the performance of UI Engine applications.

Release Information

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM Limited in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Document 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.

ARM Web Address

The ARM website is located at the following address: <http://www.arm.com>

The Mali Developer website is located at the following address: <http://www.malideveloper.com>

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give the following:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms if appropriate.

Feedback on this document

If you have any comments on or about this document, please send email to errata@arm.com giving the following:

- The document title.
- The document number.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

General suggestion for additions and improvements are also welcome.

1 THE STRUCTURE OF A UI ENGINE SCENE

A major component of the UI Engine is a simple scenegraph handler. This handles objects of type `MDE::SceneAsset`, which store graphical scenes as one or more trees of nodes. Each node may include transformations (either absolute or relative to its parent), lights, cameras, and geometric objects (with associated materials). The geometric objects are represented by `MDE:GeometryAsset` objects. To draw a scene the application must scan the tree and draw each visible geometric object in the scene using that object's `draw()` member function.

2 THE NEED FOR CULLING

The Mali GPU will filter out or overdraw faces and parts of faces that are invisible from the current camera position, therefore drawing invisible objects will make no difference to the appearance of the final images. This process is, however, inefficient for objects that are completely invisible; since the data has to be passed to the GPU before the GPU can filter it. In particular this adds to the load on the **CPU**, since passing data to the GPU requires the CPU to copy the data.

If the application can efficiently avoid drawing (cull) invisible or unimportant objects in the scene then this can often significantly improve the performance of the application.

3 CULLING TECHNIQUES

<http://www.gamedev.net/reference/articles/article1212.asp> gives a good description of a variety of culling techniques that one may use to improve graphics performance. Of these, backface culling culls individual faces; and is implemented within the Mali hardware (if it is enabled by the OpenGL ES API). The other techniques described in that page cull the scene by removing complete geometric objects from the scene, and as such can only be implemented at a higher level, where the software has access to the object structure of the whole scene.

4 BOUNDING BOXES

Most culling algorithms work by excluding objects that are completely invisible from the camera (either because they are outside the camera's frustum, or because they are completely hidden by some other object). The visibility of objects can, however, be difficult and expensive to calculate for complex objects, particularly for concave objects. One way to simplify this is to calculate a simple convex bounding box for each object that fully encloses the object. The culling algorithm can then be run on the bounding boxes. This will, occasionally, result in invisible objects remaining as candidates for drawing, but will normally remove most invisible objects.



Figure 1 A bounding box

The UI Engine (from release 2.3.1 onwards) includes a function `GeometryAsset::getBoundingBox()`. This calculates a minimal 3-D rectangular bounding box for the object in object co-ordinates. For efficiency the UI engine calculates the bounding box when the function is first called, and caches the result for later calls. It is only recalculated if the geometry of the asset is changed.

5 AN EXAMPLE OF USING BOUNDING BOXES

The Thorium example, available from <http://www.malideveloper.com> includes near plane and far plane culling (a simplified version of frustum culling). The code that implements this is in drawTree, and is shown below.

```
// Culling code does camera plane and far plane culling. Enabled by --cull option
if(cull) {
    const float tooFar_distance = 600;
    vec3 boundingBox[8];
    if(tree->data->getBatch(i).geometry->getBoundingBox(boundingBox)) {
        visible = false;
        tooFar = true;
        for(int j=0; j < 8; j++) {
            // Apply the world transformation to the bounding box
            vec4 transformedCorner = world * vec4(boundingBox[j]);
            // Convert the new position to a 3 vector
            transformedCorner = transformedCorner/transformedCorner.w;
            vec3 cornerDir = vec3(transformedCorner.x, transformedCorner.y, transformedCorner.z)
                - camPos;

            const float perpendicularDistance = cornerDir * camDir;
            // Note that near plane culling and far plane culling must be done
            // independently for each corner, otherwise we will, for example, miss objects
            // which surround the camera, with some corners behind the camera, and all the
            // others are beyond the far plane.
            if(perpendicularDistance > 0.0) {
                // If any corner is visible the object may be in front of the camera
                visible = true;
            }
            if(perpendicularDistance < tooFar_distance) {
                // If any corner is close enough to be seen then the
                // object may be close enough to be seen.
                tooFar = false;
            }
        }
    }
}
// Now draw the geometry
if(visible && !tooFar) {
    visible_count++;
    if (world.determinant() > 0) glCullFace(GL_BACK);
    else glCullFace(GL_FRONT);
    tree->data->getBatch(i).geometry->draw();
}
```

Enabling this code in the Thorium example typically approximately halves the number of objects drawn on each frame and significantly reduces CPU load. Note that it does not significantly increase GPU performance, since most of the fragments of the culled objects will be thrown away by the GPU's own culling algorithms before they reach the fragment processor, which is normally the critical path in the GPU.

6 FRUSTUM CULLING

The example above could easily be extended to simple frustum culling by calculating whether the bounding box for each object intersects the frustums. Some care is needed here, however, since the bounding box (and hence the object) may intersect the frustum even if none of its corners are within the frustum.

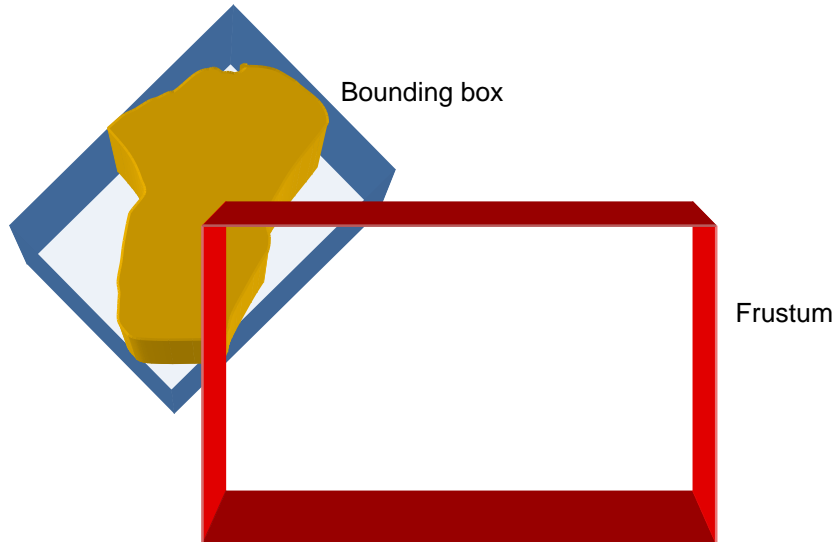


Figure 2 Bounding box corners outside frustum

For very large static scenes more complex cell based frustum culling algorithms, such as octree culling, may be faster. Such algorithms pre-organize the objects into a hierarchy of 3-D cells, and then, on each frame, exclude cells that are outside the frustum. This means that large groups of objects may be culled in single operations.

While bounding boxes may be used to speed up the sorting of objects into cells, the UI Engine contains no other direct support for such algorithms. The nodes of the scenegraph, however, can have user data attached to them using the `NodeAsset::SetUserPtr()` function. This could be used, for example, for tagging each node with a cell number.

7 OCCLUSION CULLING

Occlusion culling can be implemented at two levels:

- The Mali GPU will cull fragments are hidden by previously drawn fragments.
- An application with detailed knowledge of the structure of a scene may be able to cull, for example, objects that are hidden from the camera by walls.

Application occlusion culling is typically cell based, and typically involves considerable pre-processing of the data to, for example, calculate the set of cells potentially visible from each other cell. Once again the UI engine provides no direct support for such techniques, but the ability to attach arbitrary user data to nodes may help in the implementation of such techniques.