A Fast Culling Algorithm for 2D and 2.5D Side-scrolling Games

Frederico Mameri Rômulo Nascimento

Federal University of Uberlândia, Dept. of Computer Science, Brazil

Abstract

This paper describes a fast culling technique for 2D and 2.5D side-scrolling games, as well as for other kinds of games that share the same visual approach. In such a technique, objects can be sorted by their X coordinates and culled out based on the position of the camera.

This paper presents the algorithms and data structures needed to implement the method here described. Some problems that might occur and their solutions are also presented. Finally, we will discuss how the same technique can be used to speed up collision detection.

Keywords: 2D games, 2.5D games, viewing frustum culling, motion parallax

Authors' contact:
{fred,romulo}@comp.ufu.br

1. Introduction

In the past decade, 2D side-scrolling games have been massively phased out in favor of 3D games. However, this situation has started to change with the advent of game industry for handheld systems (such as cell phones and Pocket PC's), the revival of portable game consoles (carried out by the Nintendo DS) and the rise in the interest in the so-called retro games.

Portable devices, such as cell phones and the Nintendo DS, also feature a number of 2D and 2.5D games in its library. Surprisingly, some of the recent, 3D-optimized powerful consoles, *e.g.*, Playstation 2, also do.

Finally, 2D games use much simpler algorithms and data structures than 3D games, and are usually the starting point for someone who is beginning to learn game programming. Teaching (and learning) culling techniques in a 2D world is also much simpler than in a 3D one.

For all these reasons, a fast culling technique for 2D and 2.5D games is necessary, despite the world's tendency to overlook these types of games.

For additional information on culling techniques applied to games, [Eberly 2000; Schröcker 2001; Zerbst and Duvel] can be consulted.

1.1 Overview

Section 2 will present the general algorithm and its data structures. Section 3 will discuss some of its most important features, as well as problems and their possible solutions. Then Section 4 will show how performance can be improved for certain kinds of games and Section 5 introduces some extensions to the basic algorithm. As a final point, we draw conclusions and present ongoing and further work.

2. The algorithm

In every two-dimensional side-scroller there is a list of objects in the game (visible or not). These objects are all drawn on the same layer. That way, the set *S* of the objects in the game is an equivalence class over their Z coordinates.

2.1 The list of objects in the game

The technique proposed here consists of placing objects on a list, sorted by the X coordinate of their center point (X_{center}). This point is defined for an object as the arithmetic mean of its X_{min} and X_{max}, where X_{min} and X_{max} are the X coordinates of the most leftward and the most rightward points of the object, respectively.

Such a list should be doubly linked, and each node should contain information such as the X_{min} and X_{max} points and additional relevant information, as well as pointers to the previous and next nodes.

The list with all the objects that can be drawn (visible or not) is called the Objects list (henceforth called the ω list). Such a data structure also features two special pointers, α and β: the former points to the first object that is actually being shown on the screen and the latter points to the last object on the screen. This is the central idea of this paper and will be developed during it.

2.2 General workings

Let $Cam_{X\min}$ and $Cam_{X\max}$ be the farthest visible column of pixels to the left and to the right of the screen, respectively. Now suppose the camera has been offset by Δx pixels along the X axis. In that case, both $Cam_{X\min}$ and $Cam_{X\max}$ will be increased by Δx.

If the width of the visible part of the object farthest to the left of the screen (pointed by α) is less than or equal to Δx pixels, than that object will not be drawn

anymore (it is now out of sight). Under that circumstance, α must point to another object to maintain its property.

The algorithm consists of a pair of searches: one is stopped when the first object to be drawn is found and α is updated, the other stops when the first object not to be drawn is found and β is updated. The algorithm determines if the currently visible objects are still going to be visible in spite of the camera displacement.

First, it searches for the first object that is going to be visible, and assigns it to α. Such a search starts in the currently first visible object, checking to see if it is still going to be visible. This process only works for strictly positive values of Δx.

Then, it checks for visible objects, *i.e.*, it stops when it finds the first element not to be drawn. It then assigns its previous element to β, who now points to the last element on the offset screen.

When it comes to negative displacements, the first search starts on β and moves backwards. Likewise, the second search stops when it finds the first object not to be shown, and α is set accordingly. This second search is, of course, also done backwards.

The complete version of the Cull algorithm that deals with both positive and negative displacements can easily be constructed from the previous two algorithms by adding a initial condition that checks if Δx is positive or not.

3. Most important features

Some properties arise from the design of this algorithm. These properties will be briefly discussed in this section.

3.1 Iterative culling

Unlike traditional culling algorithms, that work on a given scene configuration, the algorithm presented here works by finding the difference from a previously culled scene.

The benefits of this kind of approach are only felt if the camera displacement was not too large. On a worst-case scenario (one in which the camera was displaced by a large value), this approach would perform as badly as any other would.

3.2 Main character

The main character of the game should not be in the ω list, for it is never going to be culled out. In fact, in most games (or game levels), it is the displacement of the main character that triggers a displacement in the camera.

3.3 Multiplayer split-screen games

It is not rare to see video games that feature a mode in which the screen is divided and each player has his/her own viewport, yet they all share the same scenario. Implementing such a feature can be easily achieved by keeping separate α and β pointers, as well as individual $Cam_{X\min}$ and $Cam_{X\max}$ variables for each player and by running the Cull algorithm for each one of them.

3.4 *Make-wide* objects

Suppose a very large object L (partially on the screen), a smaller object S (totally out of the screen), whose X_{min} and X_{center} are located between L's X_{min} and X_{center} points, and a positive displacement of the camera.

As looking for the first object not to be drawn, the Cull algorithm will detect that S is entirely out of screen, and so it will stop, making β point to the object immediately before it. L will not be drawn at all (even though it should partially be). In order to address this problem, consider the following definition.

Definition 1: An object w is wide if there is some object m so that both $X_{\min}$ and X_{center} of m lie between $X_{\min}$ and X_{center} of w or both X_{center} and $X_{\max}$ of m lie between X_{center} and $X_{\max}$ of w. In that case, m is a make-wide object.

Now that the notion of a make-wide object was presented, we will introduce four alternatives to solve the problem, and discuss the advantages and disadvantages of each one.

Multiple layers

The ω list could be split up into n smaller lists δ_k ($1 \leq k \leq n$), so that $\delta_1 \cup \delta_2 \cup \ldots \cup \delta_n = \omega$ and no wide objects exist within any list. In this case, each list δ_k would have its own α and β pointers.

An advantage of such a method is that each layer may individually give its own weight to the camera displacement (*i.e.*, each layer may multiply Δx by a scale factor). That is exceptionally useful for the implementation of motion parallaxing, which gives some layers the appearance of being farther away than others (because they move slower) and is useful for creating an illusion of depth [Hii 1997].

A disadvantage of this method is that the algorithm Cull has to be run for each layer, meaning that the overall performance will decrease.

Make-wide bit

An alternative to using multiple layers is adding a flag bit to each node in the ω list, indicating whether that object is make-wide or not.

This way, the Cull algorithm can be modified as follows: if during the search a make-wide object is found, the search continues even if it should stop. That way, wide objects that should have been detected but were not by the original algorithm will now eventually be.

The problem with this approach is that objects that are not on the screen (make-wide objects) will be located between the α and β pointers, *i.e.*, it will be considered for being drawn onto the screen.

Mixed approach

The two approaches mentioned above can be combined, so that neither the number of layers nor the number of make-wide objects is too high (meaning that neither the Cull algorithm will have to be run too many times nor the while loop will run for too long).

Delegate objects

A fast solution to this problem can be achieved by placing only the wide object in the ω list, and then have this object point to all its make-wide objects. That way, if the algorithm decides that the wide object is to be drawn, then all the make-wide objects will also be. In this technique, we called the wide object a delegate object.

4. Performance improvement

In strictly 2D games based on tile sets, it is possible to further improve the algorithm's performance by dealing with indices instead of pointers. This is possible because many games in this category split objects up into tiles instead of considering them single large objects.

The usage of tiles is memory saving. The same tile may appear many times to form a single object. Without using tiles, this information would be stored and processed repeated times. Moreover, the same tile may be simply flipped, either horizontally or vertically, to form objects (this is especially used on edges).

Another advantage of tile-based games is that it is easy to use a map editor to create levels for them.

This technique is usually implemented by considering the game level a matrix of numbers, where each number represents a tile univocally.

In order to use the algorithm, the values of $Cam_{X\min}$ and $Cam_{X\max}$ are converted into α' and β', which represent the indices of the first and the last columns to be drawn on the screen.

Notice that calculating α' and β' is much faster than running the original Cull algorithm. Once α' and β' are calculated, it is enough to draw every column c ($\alpha' \leq c \leq \beta'$) subtracted by d, where d is the number of pixels in the first column that should not be drawn.

Tile-based games can also be used in kinds of games other than side-scrollers. This technique, known as isometric projection, has been extensively used to simulate 3D games [Van Looy 2003]. That is the case of many famous commercial turn-based strategy games and ecosystem simulators.

5. Some extensions to the algorithm

In this section, we are going to discuss how the same algorithm can be used in different ways, and the necessary modifications, if needed.

5.1 Support for moving objects

So far the algorithm has only dealt with static objects, such as trees and walls. Nonetheless, computer games usually present moving objects, such as platforms and enemies that patrol a given area.

These objects are constantly moving, and so are their $X_{\min}$, X_{center} and $X_{\max}$ points. If the moving objects were simply to be placed in the ω list, then it would have to be constantly re-sorted and the Cull algorithm, run each time that happened. Clearly, this solution is not desirable.

We propose two solutions for this problem, presented next.

Roomy objects

If the object moves in a predictable, fixed way (such as platforms, or enemies that patrol a given area), that object could be considered a roomy object.

Roomy objects have their $X_{\min}$ set as the X coordinate of the most leftward point they can ever have during their path. Likewise, their $X_{\max}$ is the X coordinate of the most rightward point they can ever have during their path. The algorithm Cull is not modified.

By making an object a roomy one, it is easy to end up with a wide object. If that is the case, it must be treated as so (the techniques discussed earlier apply).

Special objects

If the moving object does so in an unpredictable way, or if its path is so big that making it a roomy object would wreck havoc the algorithm, then the object should be treated as a special object, which are always *considered* for drawing, regardless of their position along the X axis.

The disadvantage of this method is that if there are too many of these special objects, the result could be that the algorithm would be rendered useless.

5.2 Support for *collidable* objects

As the player's character will always be entirely on screen, it can only collide against objects located between the α and β pointers. That way, collision detection can be done quite fast: to check for collision detection, it is enough to check the *collidable* objects located between the α and β pointers.

Furthermore, has the list been divided into layers, the search for collision is restricted to some layers, *e.g.*, the background layers will never collide against the main character.

5.3 Support for invisible objects

Invisible objects are often used in order to trigger events in the game. This will happen whenever the main character will collide against the invisible object. The Cull algorithm is capable of handling these invisible objects without further modifications.

The same class of invisible objects that can be used to trigger actions in the game itself can also be used to spark actions in the game engine.

We are going to use invisible objects to improve the performance of the algorithm, by culling objects for a range greater than $Cam_{X\max} - Cam_{X\min}$.

That way, the Cull algorithm will need not be run every time there is a displacement in the camera, but rather, only when the main character will collide against these invisible objects.

The greater the size of the culling region, the less the Cull algorithm will be run and the more the number of objects that will be considered for drawing. Therefore, should this approach be used, one should bear in mind this tradeoff.

5.4 Adapting the algorithm for 2.5D Games

A 2.5D side-scrolling game (or 2.5D side-scroller) is a game that uses 3D polygonal meshes to render the scene, including characters, but the gameplay is like that of strictly 2D games.

3D objects will have their $X_{\min}$ and $X_{\max}$ fields set as the X coordinates of the projection on the screen plane of their points most to the left and most to the right. Likewise, $Cam_{X\min}$ and $Cam_{X\max}$ refer to values on the screen plane. We called this process *normalizing* objects to 2D.

Once all the objects are normalized, all the techniques previously mentioned apply, except for the tiles matrix.

6. Conclusion

We have presented a fast culling technique for 2D and 2D-like 3D games that also speeds up collision detection. Algorithms and data structures have been presented and problems and solutions have been discussed.

A disadvantage of this method is that it is used primarily by side-scrolling games (although some other kinds of games may find use in it, such as RPG's or ecosystems simulators). Other kinds of games with a different visual approach may find little or no use in the technique here proposed.

Ongoing and further work

This is not a finished work. We are currently researching the optimal parameters for each of the tradeoffs presented along this paper.

Next, we plan on working on a modified version of the Cull algorithm that takes into account multidimensional displacements (*i.e.*, changes in Δy and Δz are also taken into account by the algorithm), allowing a greater range of game types to benefit from it.

Acknowledgements

Professor Sandra de Amo for reviewing this paper. Anonymous reviewers for their invaluable comments.

References

EBERLY, D. H., 2000. *3D Game Engine Design : A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann.

HII, D., 1997. zLayer: simulating depth with extended parallax scrolling. *In: Proceedings of the ACM symposium on Virtual reality software and technology, 15-17 September 1997 Lausanne, Switzerland*. New York: ACM Press, 65-69.

SCHRÖCKER, G., 2001. Visibility Culling for Game Applications [online] Graz University of Technology. Available from: www.schroecker.info/download/pvs.pdf [Acessed August 18 2006].

VAN LOOY, J., 2003. Interactivity and signification in Head Over Heels [online] *The International Journal of Computer Game Research, December 2003*. Available from http://gamestudies.org/0302/vanlooy/ [Accessed August 24 2006].

ZERBST, S. AND DUVEL, O., 2004. *3D Game Engine Programming*. Course Technology PTR, 1st edition.