Ibero-American Symposium on Computer Graphics - SIACG (2006)
P. Brunet, N. Correia, and G. Baranoski (Editors)

Efficient Conservative Collision Detection for Populated Virtual Worlds

A. Ramires Fernandes[1] and L. Deusdado[2]
[1]CCTC, Universidade do Minho, Portugal
[2]Instituto Politécnico de Bragança, Portugal

Abstract

Large virtual worlds, with considerable level of detail are starting to emerge everywhere, from large areas of actual cities to archaeological detailed reconstructions of sites. Populating a virtual world adds an extra touch to the visualization of these worlds, but unfortunately it also brings an extra burden to the system. Several tasks are required when adding animated characters to a virtual world, such as collision detection, path planning and other AI algorithms, rendering of dynamic geometry, amongst others. In here a method for efficient and scalable conservative collision detection is presented, that is able to deal with large scenes and thousands of avatars. This method does not perform exact collision detection, hence it is conservative. The method is suitable as a basis for path planning algorithms and other AI algorithms where an avatar is often regarded as 'something' that can be bounded by a cylinder, or a box. The algorithm is capable of dealing with arbitrarily complex 3D worlds, and does not require any a priori knowledge of the geometry.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Three-Dimensional Graphics and Realism]: Virtual RealityAnimation; I.3.5 [Computational Geometry and Object Modeling]: Geometric Algorithms;

1. Introduction

With the advent of more and more powerful shader programming enabled hardware, capable of rendering millions of triangles, large projects are starting to emerge everywhere, virtualizing, with considerable level of detail, large portions of cities, ancient archaeological sites, or architectural projects that are yet to be realized.

These 3D worlds becomes more interesting and "realistic" as soon as they are populated with virtual characters, or avatars. Having these characters inside the 3D world provides another clue for apprehending the context of the virtual world, an interesting example is the usage of avatars to provide a sense of scale.

Visualization of such projects in real time, requires one to use a set of performance related techniques to achieve interactive frame rates. Regardless of how powerful the graphics hardware becomes, the quest for visual realism (for instance the inclusion of quasi-global illumination models that work in real time), means that a large chunk of both CPU and GPU is required.

In addition to the visualization performance issue, populated virtual worlds bring yet another burden on the system: that of collision detection between the avatars and the world, as well as between the avatars themselves.

Collision detection in populated worlds can be seen in two different perspectives: exact and conservative collision detection. Imagine an avatar walking on a city garden. Conservative collision detection can be used while the avatar is walking on the garden. When the avatar reaches a newspaper stand and it reaches for a newspaper or any other item, then exact collision is required.

In here an efficient method to perform conservative collision detection between avatars and a 3D world is presented. The method makes no assumptions on the 3D world, which can be a 'soup of unrelated polygons as far as modeling is concerned. The method is able to deal with arbitrarily complex worlds and different avatar sizes without compromising performance scalability.

The paper is structured as follows: section 2 provides an overview of previous work in the area, focusing on methods that are designed and tested with avatars, and the works that we're the basis for the method presented in here; section 3 details the method, namely the preprocessing stage and the collision detection algorithm; section 4 shows examples of the application of the method, including the time required for both pre-processing and collision detection with very large numbers of avatars. Finally conclusions and future work are presented in section 5.

2. Background

Collision detection from a geometrical point of view, i.e. between generic geometrical objects, has been presented based on different approaches, mostly supported by an hierarchal structure: bounding boxes [GLM96], sphere trees [Hub93], BSPs [NAT90], and octrees [Sam90].

As mentioned in the introduction, collision detection can be considered under two different perspectives: exact and conservative collision. Exact collision is far more precise, as the designation points out. The ability to detect collisions precisely is also far more computationally intensive.

Examples of works that perform exact collision detection with hierarchical bounding volumes can be seen in [CMM95], [WLML99] and [RKL*04]. A different approach is taken in [VP05] where the avatar casts rays into the environment to detect obstructed paths. In [GRLM03] determine potentically colliding sets with visibility queries and use the information to perform exact collision detection. Collision detection is performed in [WS04] based on depth maps taken from a frustum that encapsulates the movement of the avatar. Interference detection is the term used in [KP03] where a method inspired in shadow volumes is described. Yet another related research area is collision detection of an avatar and its clothes [SK04].

Conservative collision detection is required when an avatar is moving in a virtual world. In this case two pieces of information are required: where are the feet of the avatar standing, and can it move forward without colliding. Collision in this latter sense takes into account if the avatar is able to walk over, or jump down an obstacle. Therefore a 2 meter wall is a collidable object, but a fence 10 cm above ground is certainly not an obstacle, assuming an avatar with human proportions.

Under simple worlds, terrain following techniques can provide the height at which the avatar should be placed, as long as the graphical primitives that make up the terrain are clearly identifiable. To obtain collision information works such as [Ste97] and [TC00] have been proposed previously. Both deal with worlds that are planar in the sense that for a particular (X,Z) position there is only a single Y value that is suitable for the avatar, assuming Y as being the vertical axis.

The work in [Ste97] uses a BSP approach where the world is decomposed in cells linked along the edges. The method is dependent on the number of edges, although it can perform incrementally dividing as the avatar moves into unchartered areas.

3D space discretization was proposed in [BT95] and [BT98]. The process involves dividing the world in thin horizontal slices, where each slice contains a grid. For each slice the geometry contained in the slice is drawn, and the grid cells without geometry are empty world cells where the avatar can potentially navigate. The resolution of this method is determined by the thickness of the slices, and the grid cell size. When considering a non flat world, for instance with ramps, or non flat terrain, the number of slices must be very large and grid cell size must be very small to capture the heights at which the avatar travels. Furthermore to determine if an avatar can move from point A to point B, a large number of cells must be tested for emptiness.

In [TC00] a method to automatically extract heights and collision detection information from an arbitrary 3D world is proposed. The heights are found by computing a depth map taken with an orthographic camera, vertically looking down on the scene. The depth map is rendered and the heights are then extracted from the depth map. This technique is a very simple way of discretization of a 3D world for the purpose of collision detection. Collision detection of an avatar against the world is performed by checking the grid cells that the avatar uses in its movement per frame. If all cells are within reach of an avatar, i.e. if all the cells have the same height, or if the difference in heights is less than what the avatar can climb, then there is no collision. Collisions occur when the avatar tries to access a grid cell that is at a height that is unreachable to the avatar because either the avatar can't climb, or because the avatar can't jump, the height difference. This technique was used in an agent behavior simulator described in [TLCC01].

However when one considers a world with multi-levels, for instance a bridge that the avatar could go over or under it, or a building with many floors, the technique by [TC00] is only capable of traveling on top of the bridge or the top of the building, since these are what is rendered on the final depth map.

Both [BT98] and [TC00] methods were the main inspiration for the method described in here. The goal is to extend the techniques described above to multi-level 3D scenes, using height maps, and allowing the avatar to go under the bridge, and on top of the bridge, or to navigate in the floors of a building. The multi-level method works with arbitrarily complex worlds, with theoretically unlimited number of levels, and it scales linearly with the number of avatars.

3. Multi-Level Collision Detection

The method presented in here provides efficient collision detection in multi-level 3D virtual worlds. An example of such virtual environment can be seen in fig. 1. In this world the avatar can navigate in both 3 floors, climb the ramps and other small obstacles. It must detect collisions with the cars, pillars, and other objects in the scene. It must also not jump down from a floor.

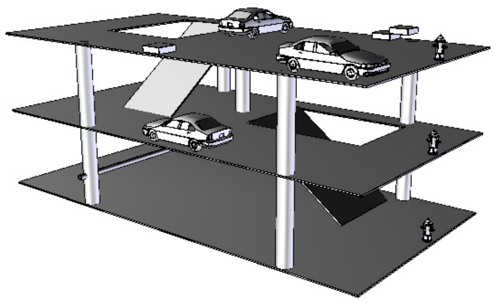

Figure 1: *Simple Multi-Level Virtual World*

The goal of the method is to provide an efficient way of controlling the avatar position, while at the same time preventing collisions with the world objects. It is assumed that the height of an avatar, *avatarHeight*, is known.

The method starts by automatically extracting information of the 3D world to determine the areas where the avatar can go and at what height the avatar should be placed when moving.

This is achieved by slicing the world with horizontal planes. For each slice, the height at which the slice was taken, as well as the height map obtained at that slice is kept. The slicing process takes into consideration the value of *avatarHeight* to decide at which height the next slice will be taken. The slicing process may generate a large number of slices because it is assumed that no a priori knowledge of the virtual world is available. However, only a few slices are actually required for navigation, so the memory footprint is kept under control. This process is detailed in sub-section 3.2.

This preprocess stage is reasonably fast since the most computationally intensive operation is rendering the depth map. Rendering the depth maps does not require shaders, lighting, and other lighting effects that slow down rendering. Furthermore, only for the first slice is the whole scene rendered. As the height at which the slices are taken decreases, less and less geometry is involved, hence the final slices should be much faster then the initial ones.

When the virtual world is being visualized, after the preprocessing stage is concluded, the slices are used to determine two important pieces of information:

- The height at which the feet of the avatar should be placed
- The free space on the areas where the avatar wants to move to

This process is also very simple from a computational point of view and it amounts to a few lookups in the slices that were stored in the preprocessing stage. The simplicity of the process allows it to perform conservative collision detection with thousands of moving avatars in an arbitrarily complex virtual world. The runtime step is detailed in sub-section 3.1.

3.1. Runtime stage: collision detection

Assume that the preprocessing stage has computed two slices for the virtual world in figure 2. The bold vertical lines represent the near planes used to render the depth maps, and the legend to these lines indicates the height at which they were positioned. In this case the first slice is taken with the near plane set at Y = 12,6, and the second slice is taken with Y = 5,9. The boxes with the numbers above each slice line, represent the pixels in the height map and the numbers indicate the height recorded.

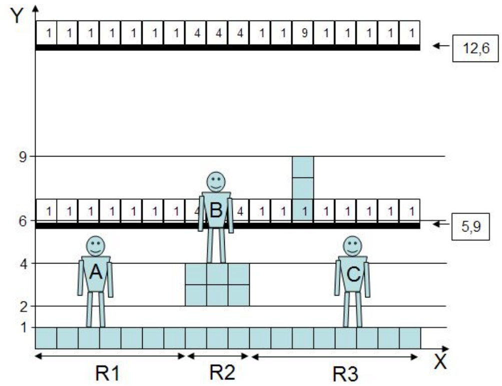

Figure 2: *Sliced World with Avatars*

An avatar has three parameters: *width*, *height* and *step*. The first two relate to its dimensions, and the latter indicates how much the avatar can jump both up and down. Assuming that an avatar has a step of less than 3 units, then the movement of the avatars is limited to the regions depicted in figure 2, for instance avatar A can only move in region R1.

This information can be extracted from the slices in figure 2. An avatar will read the information of the slice that is immediately above its head. So avatars A and C will read the information on the lower slice, and avatar B will read the information on the upper slice. If avatar B tries to move to region R1, it will find that it must move from a height of 4 units to a height of 1 unit. Assuming that the avatar's step is less than 3 units, the movement would be classified as illegal. Similarly avatars A and C can't move to region R2.

The world has been discretized in cells on the X axis (in the 3D case the discretization would be in the XZ plane). An avatar stands in one of those cells at a particular height *h*, this may be for instance the height of the top of its head. (avatars A and C would have h = 5.5, and avatar B would have h = 8.5).

When an avatar wants to move to a neighboring cell, the motion is then decomposed into a vertical motion followed by an horizontal motion. First it is necessary to check if the avatar's step is higher than the height difference between the current cell and the new cell. Assuming that the avatar step is smaller than its height, this requires only checking the magnitude of the vertical movement. Assuming that this magnitude is not superior to the avatar's step then it is necessary to check if there is free space for the avatar to move.

If after the vertical movement the avatar's head new height is still below the original slice, then the motion is legal. If the avatar is moving up and its head is now above the original slice, then it is necessary to use a new slice, more precisely the slice above its head after moving up to validate the movement.

Algorithm 1 describes this process in detail.

```
boolean move(A,B) {
·  hA = slice[i][A] + avatarHeight;
·  hb = slice[i][B] + avatarHeight;
·  if (|hA - hB| < avatarStep) {
   ·  if (hb > sliceHeight[i]) {
      ·  // find the slice above the avatars head after the vertical
         movement
      ·  j = i;
      ·  while (hb > sliceHeight[++j]);
      ·  if (slice[i][A] == slice[j][A])
         ◇  return(LEGAL);
      ·  else // there is something preventing the vertical move-
         ment
         ◇  return(ILEGAL);
   ·  else
      ·  return(LEGAL);
·  }
·  else
   ·  return(ILEGAL);
}
```

Algorithm 1: Runtime algorithm to evaluate whether an avatar movement is legal or illegal

Collision detection amongst avatars is also solved using a similar strategy. An extra bit is kept for each cell that states its occupancy status. The bit must be checked prior to moving the avatar to check for avatar-avatar collision.

3.2. Preprocessing stage: slicing the virtual world

This section details the preprocessing stage of the method and presents several examples that illustrate common situations.

Initially an axis aligned bounding box of the virtual world is computed. This process can be performed at almost no extra cost when the model is loaded. The maximum and minimum values on each axis are stored as *maxX,minX,maxY,minY,maxZ,minZ*. An orthographic camera is then placed on top of the world, looking down the Y axis such that the view frustum includes the full bounding box. The near plane is set above *maxY* value recorded, and the far plane is set bellow *minY* value (the darkest planes in fig. 3 represent the near and far planes).

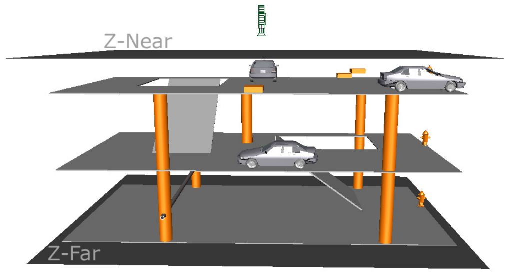

Figure 3: *Simple Multi-Level Virtual World*

To simplify the presentation of the method, and without loss of generality, the diagrams will be presented in 2D, representing sections in the plane XY from the virtual world, see figure 4.

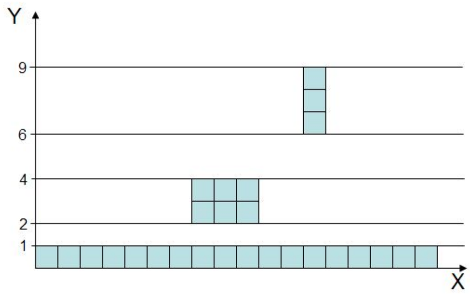

Figure 4: *2D Simplification of Virtual World*

The near plane is initially placed at an height defined in eq 1.

$$hNearPlane = maxY + avatarHeight + resolution \quad (1)$$

where *resolution* indicates the maximum vertical error in the discretization of the virtual world for collision detection purposes. This error does not influence the ability of the avatar to keep its feet on the ground, as that information is stored in the height map. This error implies that the avatar may have a space over its head that will in the worst case be the full value of *resolution*.

The far plane is constant throughout the process and it is set at an height defined in eq 2.

$$hFarPlane = minY - resolution \quad (2)$$

The depth map is then rendered and it is processed in order to obtain a height map. Assuming an avatar with an height of 4.5 units, and a resolution of 0.1 units the first slice would be taken at Y = 12.6. The maximum registered height, according to figure 4 would be 9 units.

Let *max(i)* be the maximum height recorded in slice *i*. The next slice, slice *i+1*, is then taken at a height as defined in eq. 3.

$$nextSliceHeight = max(i) - resolution \qquad (3)$$

The first two slices are depicted in figure 5.

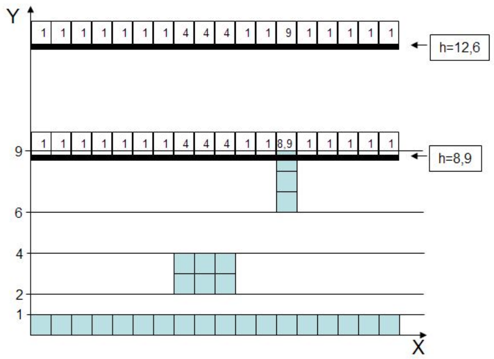

Figure 5: *First and second slices*

The third slice, according to eq. 3, should be taken at height Y = 8,8. In fact slices will be taken at intervals defined by the parameter *resolution* until Y = 5,9, see figure 6.

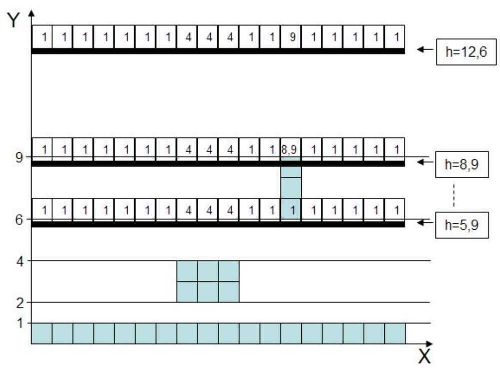

Figure 6: *Slices up to 5.9 units*

The maximum value in the last slice, Y=5.9, is 4, and since the minimum avatar height as been assumed to be 4.5 there is no need to get any more slices. Assuming that *resolution* is set at 0.1, total of 32 slices are computed in the process, however only two are required to perform collision detection, namely the first and last ones. All other slices do not carry any further information relevant to collision detection.

A slice is used by an avatar to perform collision detection when it is the closest slice above its head. Hence, considering a particular slice, if a cell has a recorded height such that the difference between the height at which the slice was taken and the height recorded is less than the avatars minimum height, the cell will never be used for testing. This is the case in the second slice in figure 6, where the recorded height is actually equal to the slice height. All other values of the second slice are equal to the corresponding values of the first slice, therefore the second slice does not carry any new information and can be dismissed. This reasoning leaves us with 2 useful slices in figure 6, the one taken at a height of 12,6 units and the one taken at 5,9 units.

The example in figure 7 shows another case where slices can be dismissed. Consider the three slices present in figure 8. The slice taken at 7,9 units of height can be dismissed because all its information is present in the first and third slices. The red crosses in each slice indicate the cells that will never be used for collision detection because the avatar will have its head above the slice in that particular position.

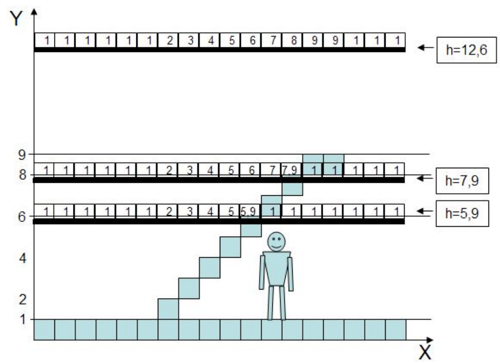

Figure 7: *Example with stairs*

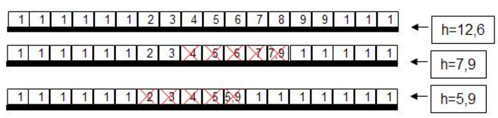

Figure 8: *Slices from stairs example*

So far two situations where a slice can be dismissed have been identified. The algorithm to detect this situations and dismiss the slices is presented in algorithm 2. The algorithm is called each time a new slice is computed, and checks if the new slice contains any new relevant information (example from figure 6). If it does not it dismisses the slice. If the slice contains new information, then the algorithm checks if the previous slice should be replaced by the new slice (example from figure 7).

```
Let slices be an array of slices taken so far.
Let sliceHeight be an array of the heights the slices we're taken
Let avatarHeight be the height of the avatar
define INVALID as being below minY
testSlice(i) {
·  auxSlice = new Slice()
·  countEqual = 0
·  countUseless = 0
·  for every cell a in slice[i] {
   ·  if (sliceHeight[i] - slice[i][a] < avatarHeight)
      ·  countUseless++
   ·  else if (slice[i][a] == slice[i-1][a]) {
      ·  auxSlice[a] = INVALID
      ·  countEqual++
   ·  }
·  }
·  if (countEqual + countUseless == number of cells in slice)
   ·  return(DISMISS_CURRENT_SLICE)
·  // are there more than two slices?
·  if (i > 1) {
   ·  for every cell a in slice[i] {
      ·  if ((auxSlice[a] == INVALID) || (slice[i-2][a] == slice[i-
         1][a]) || (sliceHeight[i-1] - slice[i-1][a] < avatarHeight))
         ◇  count++;
   ·  }
   ·  if (count == number of cells in slice)
      ·  return(REPLACE_PREVIOUS_SLICE)
·  }
·  return(KEEP_SLICES);
}
```

Algorithm 2: function *testSlice*

In algorithm 3 the full process of slicing is detailed, assuming that a bounding box has been computed and Y varies between *maxY* and *minY*.

3.3. Implementation Details

As mentioned before, the method renders a depth map and then transforms it into a height map. In recent hardware, with floating point buffers, the scene could be rendered with appropriate shaders to perform these two steps in one go.

Rendering the depth map is conceptually a sound idea to obtain a height map, however, in practice, some problems arise. These problems are related to the way graphic primitives are used to model a scene and the Z buffer resolution.

```
sliceWorld(maxY, minY) {
·  sliceHeight[0] = maxY + avatarHeight + resolution
·  slice[0] = ComputeSlice(sliceHeight[0])
·  max = maximum(slice[0])
·  i = 1
·  While (max > minY + avatarHeight) {
   ·  sliceHeight[i] = max - resolution
   ·  slice[i] = ComputeSlice(sliceHeight[i])
   ·  max = maximum(slice[i])
   ·  test = testSlice[i]
   ·  if (test == KEEP_SLICES)
      ·  i++
   ·  else if (test == REPLACE_PREVIOUS_SLICE) {
      ·  slice[i-1] = slice[i]
      ·  sliceHeight[i-1] = sliceHeight[i]
   ·  }
·  }
}
```

Algorithm 3: The slicing algorithm

A wall may be commonly modeled using vertical triangles. These triangles are perpendicular to the near and far planes, and hence have zero projection area. Normally these polygons are not rendered, and therefore their height is not recorded in the z-buffer. Consider the 3D scene depicted in figure 1. When taking a slice below the third level, different results are obtained if lines are drawn on top of the polygons. Figure 9 (top) shows the depth map (darker means higher) obtained when rendering the polygons on the scene using fill mode, and figure 9 (bottom) shows the depth map when lines are drawn on top of the polygons (note: these figures have been contrast enhanced to illustrate more clearly the differences between them).

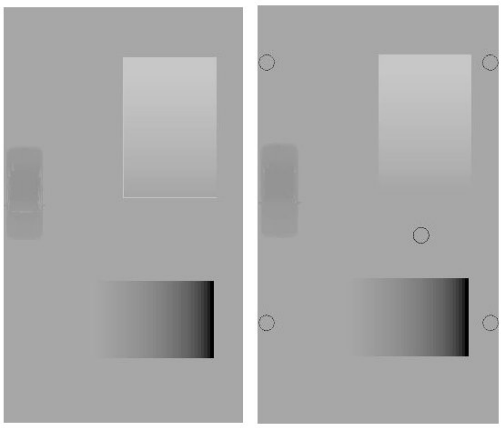

Figure 9: *Depth Maps (left: fill polygon mode; right:lines superimposed on top of the polygons)*

The differences in figure 9 are in the columns. These are represented by vertical polygons that are not rendered when using polygon fill mode. Adding the lines on top of the polygons produces the contours of the columns which is a step in the right direction. However when converting the depth map to heights, we realized that there was a mismatch between the real height and the height map values for the pixels relating to the columns.

Regarding the z-buffer resolution, this is typically limited to 24 or 32 bits and its non linear, hence some discrepancies may appear when reading depths taken with different near planes. For instance a 3D point measured with different near planes will probably yield different depths in the Z buffer. If considering a 24 bit z-buffer, this difference is in general very small, and can be accounted by using a threshold when testing numbers for equality.

Resorting to clip planes also allows us to deal with another Z buffer issue: the Z buffer is not linear, using more precision for areas near the clip plane that for areas close to the far plane. When one considers the depth maps for two slices taken with two different Z-near planes, the depth recorded values, for the same pixels, may be different in the two slices due to the non-linearity of the Z buffer. By using a clip plane, we are able to keep the Z near and Z far planes fixed and vary only the clip plane, hence guaranteeing that for the same pixels, the heights recorded are the same for all slices.

A more relevant issue has to do with the precision of the height maps, and consequently, the memory required per slice. If one considers a world where a unit corresponds to a meter, then a 16-bit height map would allows us to deal with scenes up to 65.536 meters with a height error of less than one millimeter. Or if one can be bore tolerant then one could go up to 655.36 meters with a height error of at most one centimeter. This is enough for the currently tallest building in the world: the Taipei Tower 101, with 509 meters.

However an error of one centimeter may be excessive in some situations. A possible solution is to store height differences, or depths in the slices, instead of the actual height. The height at which a slice was taken can use as much precision as required, and the slice values would store the depths. In runtime only an extra subtraction would be required. The method is therefore not limited by precision issues, even considering memory saving features, such as storing the heights/depths with 16-bit precision.

There may be a negative implication when using limited precision. This will happen when the actual height/depth requires more than the available precision. These values should be considered invalid and extra slices should be added to guarantee that in every situation where there is an area that can be navigated by an avatar, there is a slice at an appropriate height/depth.

If the height variation is small enough so that precision is not an issue, than one bit can be used to indicate if the cell is occupied by an avatar, otherwise an array of bits should be considered for avatar-avatar collision detection.

4. Experiments

Tests have been performed in the garage scene (see figure 1) to illustrate the concept, and in the powerplant model (model available at http://www.cs.unc.edu/ geom/Powerplant/) to show its applicability to very large scenes (by today's standards).

The slices taken for the garage scene are shown in figure 11. The processing time required to slice the scene (n slices were generated) and to eliminate the unnecessary slices was less than one second.

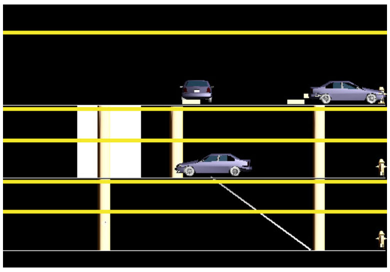

Figure 10: *Slices obtained for the garage scene*

Tests were also performed to see the performance in the collision detection phase. The garage scene took 10 milliseconds to render. The results are presented in table 1. For reference purposes the time taken to draw an avatar (represented graphically by a box) is also presented. The number of avatars tested ranged from 500 to a million avatars. As can be seen the method scales linearly with the number of avatars as expected. Also note that the time to move the avatar also includes deciding a new direction in case of collision, and testing the new direction. The algorithm to decide a new direction when a collision occurs is simply a random choice of left or right.

Nr. of Avatars	draw avatars	move avatars
500	2	2
1000	5	4
1500	7	6
2000	9	8
5000	23	20
10000	48	39
100000	467	395
1000000	4780	3990

Table 1: *Performance results for collision detection on the garage scene (time in milliseconds)*

Other scenes were tested for the number of slices, namely a curch buiding , and the powerplant http://www.cs.unc.edu/~geom/Powerplant/ from the *Walkthru Project* at Stanford University. Table 2 shows the number of slices taken and kept for each world. All tests assumed a avatar with the equivalent height of a 1.75 meters, and a resolution of 10 cm.

Scene	taken	kept
cube	42	2
garage	86	5
church	159	7
powerplant	835	85

Table 2: *Slices*

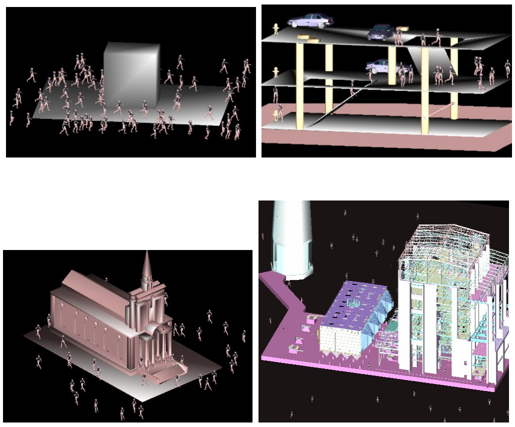

Figure 11: *Tested Environments: cube, garage, church and powerplant (clockwise from top left)*

5. Conclusions and Future Work

A method for Multi-Level collision detection for arbitrarily complex 3D worlds was presented. The method is able to detect collisions between the avatars and the virtual world, as well as avatar-avatar collisions, coping with avatars with different heights.

The tests show that collision detection in complex environments in a multi-level 3D world can be performed with very large number of avatars at interactive rates, and that the run-time performance is not significantly influenced by the complexity of the 3D world. A more complex world may require more slices, and potentially more collision processing time, but the tests show that even when considering very complex worlds this does not represent a significant penalty in performance.

Although the memory footprint is perfectly acceptable for the examples tested in here. The number of slices taken is kept to a minimum by testing the usefulness of each slice. Nevertheless some occasions may rise where memory usage is a concern, hence a future direction is to explore algorithms that deal with sparse matrices, and evaluate the trade off between memory consumption and performance. Another possibility is to evaluate the feasibility of using an out-of-core algorithm to store and retrieve the matrices. Yet another avenue of research that may bear fruits is the exploration of the information in the slices for real-time path planning.

References

[BT95] BANDI S., THALMANN D.: An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies. In *Computer Graphics Forum* (1995), Post F., Göbel M., (Eds.), vol. 14(3), Blackwell Publishers, pp. 259–270.

[BT98] BANDI S., THALMANN D.: Space discretization for efficient human navigation. In *Proceedings of the Eurographics Conference, Computer Graphics Forum* (1998), vol. 17, pp. 295–270.

[CMM95] COHEN J., M.LIN, M.PONAMGI: I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the ACM Interactive 3D Graphics Conference* (1995), pp. 189–196.

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: Obb-tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH Conference Proceedings* (1996), pp. 171–180.

[GRLM03] GOVINDARAJU N., REDON S., LIN M., MANOCHA D.: Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.* (2003).

[Hub93] HUBBARD P. M.: Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers on Virtual Reality* (1993).

[KP03] KNOTT D., PAI D.: Cinder: Collision and interference detection in real–time using graphics hardware. In *Proc. of Graphics Interface '03, 2003.* (2003).

[NAT90] NAYLOR B., AMANATIDES J., THIBAULT W.: Merging bsp trees yields polyhedral set operations. In *ACM Computer Graphics* (1990), vol. 24(4), pp. 115–124.

[RKL*04] REDON S., KIM Y., LIN M., MANOCHA D., TEMPLEMAN J.: Interactive and continuous collision detection for avatars in virtual environments. In *Proceedings of IEEE International Conference on Virtual Reality* (2004).

[Sam90] SAMET H.: The design and analysis of spatial data structures.

[SK04] S. KIMMERLE MATTHIEU NESME F. F.: Hierarchy accelerated stochastic collision detection. In *Vision, Modeling, and Visualization* (2004).

[Ste97] STEED A.: Efficient navigation around complex virtual environments. In *VRST* (1997), pp. 173–180.

[TC00] TECCHIA F., CHRYSANTHOU Y.: Real-time rendering of densely populated urban environments. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (2000), pp. 83–88.

[TLCC01] TECCHIA F., LOSCOS C., CONROY R., CHRYSANTHOU Y.: Agent behaviour simulator (abs): A platform for urban behaviour development. In *Games Technology 2001(GTEC 2001)* (2001).

[VP05] VOSINAKIS S., PANAYIOTOPOULOS T.: A tool for constructing 3d environments with virtual agents. *Multimedia Tools Applications 25*, 2 (2005), 253–279.

[WLML99] WILSON A., LARSEN E., MANOCHA D., LIN M. C.: Partitioning and handling massive models for interactive collision detection. In *Proceedings of the Eurographics Conference, Computer Graphics Forum* (1999), vol. 18(3).

[WS04] WINTER M., STAMMINGER M.: Depth-buffer based navigation. In *Vision, Modeling, and Visualization Conference Proceedings (VMV)* (2004), pp. 271–278.