Image Based Collision Detection

Rafael Kuffner dos Anjos

October 25, 2011

Abstract: This work explores an alternative approach to the problem of collision detection using images to represent complex environments and buildings derived from laser scan, creating several 2.5+D maps that together represent a 3D view of the world.

Keywords: Collision Detection, Image-based, Point-Cloud, Interactive Applications, Polygonal Oversampling.

1 Introduction

1.1 Problem and motivation

Collision detection is normally a bottleneck in the visualization and interaction process, as collisions need to be checked at each frame. Traditionally, the more complicated and crowded is our scene, the more calculations need to be done, bringing our frame-rate down. Therefore the optimization of this process, gaining speed without losing quality in the simulation, is something that has been researched for years.

Different techniques and approaches such as bounding volumes, distance fields, and other auxiliary data structures that group object features together to diminish the number of useless testings have been developed, and showed good performance.

Invariantly, in more complex or crowded scenes, or on a situation where one does not have enough processing power available, one quickly reaches a performance bottleneck while trying to keep the realism of the simulation. Also, in scenarios that use a different object representation such as a point cloud, we can’t rely on object topology information. The classical approaches either won’t work, or will have to heavily adapt to this specific scenario, tailoring its optimizations to a point cloud representation.

Using images as an alternative way of executing the task of collision detection might just be the answer. Image-based techniques can have their precision easily controlled by the resolution of the used images, and the algorithms are completely independent of the object’s topology. It doesn’t matter whether we have a pointy object, a round one, or even a point cloud, as all we’re dealing with is the object’s image representation. Being a scalable and promising technique, Image-based collision detection seems to be a plausible alternative to the classical approaches.

Our approach focuses in a virtual reality navigation scenario, where we’re representing data coming from the real world via devices such as laser scans, which present us with enormous point clouds. Also, the hardware available on our hands might not fit the basic requirements for most algorithms and techniques, a situation that commonly will happen in tourism hotspots, museums, or other places where we want ordinary people to interact with the system.

The main contribution of our research is a new 3D world representation for environments and buildings completely image-based with useful information for collision-detection queries. Slicing the structure along the Z axis (Figure 1), we create a discrete set of images containing height information about the surface, and possible collidable frontiers. It’s a flexible format, that is able to represent either point clouds or polygonal models. This representation allow us to perform collision detection with user chosen
2 RELATED WORK

The problem of collision detection is present in varied areas of research and applications, each of them having different concerns and desired results. This necessity has given birth to several techniques that try to deliver these results using varied approaches.

The goal of collision avoidance is to predict an upcoming collision, and use this information to prevent it from happening. It is used mostly in artificial intelligence to control intelligent agents or robots.

The BOIDS framework [24] uses simple circular spheres are used to detect obstacles in the vicinity. On Ondrej et. al [23] work, a more complex technique is presented that simulates the human vision to detect obstacles on the agent’s path. Loscos et. al [20] represent the environment as patches to where the agent can or cannot go according to it’s occupation.

Feature based algorithms use the vertices, edges and faces (so called features) of the object’s boundary to perform geometric computations to determine if a pair of polyhedra intersect and possibly calculate the distance between them. Famous examples are the Lin-Canny algorithm[19] and it’s more recent related algorithm, V-Clip[21], that keeps track of the closest features between two objects, deriving both the separation distance, and the vertices that have possibly already penetrated the other object.

By far the most popular class of collision detection algorithms, BVHs work by dividing the objects in smaller primitives contained by it, until a certain leaf criteria is achieved (top-down approach) or starting on the smallest primitives possible, grouping up until a single volume is achieved (bottom-up). Each primitive is enveloped by a particular bounding volume shape such as Spheres [15], Oriented Bounding Boxes (OBB) [12], or Axis Aligned Bounding boxes (AABB) [18] [11] [28], each of them having their advantages over the others; Spheres are easier to fit, OBBS have faster pruning capabilities, and AABBS are quicker to update, therefore being very popular in deformable body simulation.

When checking for collisions between two objects, the hierarchy is traversed top-down, testing only elements which have parents that have collided with another BV, avoiding useless calculations. Different tree traversing and creation techniques [18] [12] have been developed to optimize these expensive operations, taking into account each specific kind of application.

Stochastic algorithms that try to give a faster but less exact answer have been developed, giving the developer the option to "buy" exactness in collisions with computing power. The technique based on Randomly selected Primitives, selects random pairs of features that are probable to collide, and calculates the distance between them. The local minima is kept for the next step and calculations are once again made. The exact collision pairs are derived with Lin-Canny [19] feature based algorithm. With a similar idea, Kimmerle et. al [25] have applied BVH’s with lazy hierarchy updates and stochastic techniques to deformable objects and cloth, where not every bounding box is verified for collision, but it has a certain probability.

Image-based algorithms commonly work with the projection of the objects, opposed to the previous techniques that work in object space. RECODE [6] and several other works [17] [22], [5] take advantage of the stencil buffer and perform collision detection on it by using objects coordinates as masks, and thus detecting possible penetrations.

CULLIDE [13] uses occlusion queries only to detect potentially colliding objects, and then triangle intersubsection is made on the CPU. They render the bounding volumes of the objects in normal and reverse list storage order, and remove the objects that are fully visible in both passes, meaning they are not involved in any collision.

Heidelberger et. al [14] uses simple AABB’s as bounding volumes for the objects in the scene. Potentially colliding objects are detected, and a LDI (Layered Depth Image [26]) of the intersubsection volume between the two
Table 1: Coarse granularity in the compromise between precision and efficiency.

<table>
<thead>
<tr>
<th>Feature based</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>BVH</td>
<td>Popular implementation, relatively scalable, reliable</td>
<td>Compromise between precision and efficiency hard to obtain, requires input data to be properly segmented</td>
</tr>
<tr>
<td>Stochastic</td>
<td>Fast and adaptable</td>
<td>Not completely reliable, shares all of BVH’S issues.</td>
</tr>
<tr>
<td>Image-based</td>
<td>Scalable and precise, topology independent, works with unstructured polygon soups</td>
<td>Requires specific graphic card capabilities</td>
</tr>
</tbody>
</table>

objects is created. That is, a volumetric representation of an object across a chosen axis. At each rendering step, as a polygon is projected into the LDI, the size of the intersubsection volume is computed. Faure et. al [10] [3] addresses not only collision detection, but also it’s response by using this same principle.

Regarding collision detection on point clouds, algorithms using feature based techniques, bounding volumes, and spatial subdivision have been developed. Similar to the idea presented in SWIFT [8], Klein and Zachmann [16] create bounding volumes on groups of points so collision detection can be normally applied. Figueiredo et. al [11] uses spatial subdivision to group points in the same voxel, and BVHs to perform collision detection.

In Table 1 we have a quick comparison of the referred techniques, where we can see that image-based algorithms are the ones more fitted to our scenario. For further information on collision detection and avoidance techniques we suggest the following surveys: [4] [9] [27].

3 Implementation

Image-based algorithms that have been presented in the community ([13] [6] [17] [7] [3] [10]) perform very well in various kinds of scenarios, but some features of our set scenario (described on section 1.1) make them hard or impossible to be applied (e.g. our data is unstructured, not all objects are closed or convex).

Our work extends the idea of a 2.5D map presented on the work of Loscos et. al [20] combining it to a a height-map, where the pixel color not only represents the height on that point, but also contains obstacle information, while overcoming the limitations of only supporting single floor environments. Instead of having just one height map, we create a series of special maps along intervals sized $\sigma$ on the $z$ axis, thus enabling the storage of more than a single $z$ value for each $(x, y)$ pair. Using the color of each pixel as a representation of a voxel, we write height information on the red channel, and identify obstacles on the blue channel. By adding these variables, we can determine not only the height where the avatar should be standing, but also if he is colliding with any obstacle in several different heights.

Figure 1: Slices creation process, and camera variables
3 IMPLEMENTATION

3.1 Slices creation

The creation of this representation is executed in a pre-processing stage, divided in several steps (Figure 2) that must be performed from the input of the model until the actual rendering to create the snapshots that will be the used as collision maps.

**Algorithm 3.1 Slices creation**

```plaintext
for z = z₀ \rightarrow z = z_{\text{max}} do
  \text{z}_{\text{cam}} \leftarrow z₀ + \sigma
  \text{near}_{\text{cam}} \leftarrow 0
  \text{far}_{\text{cam}} \leftarrow \sigma
  \text{left}_{\text{cam}} \leftarrow \max(\text{z}_{\text{max}}, \text{x}_{\text{max}})
  \text{right}_{\text{cam}} \leftarrow \min(\text{z}_{\text{min}}, \text{x}_{\text{min}})
  \text{bottom}_{\text{cam}} \leftarrow \max(\text{z}_{\text{max}}, \text{x}_{\text{max}})
  \text{top}_{\text{cam}} \leftarrow \min(\text{z}_{\text{min}}, \text{x}_{\text{min}})
  \text{render and snapshot}
end for
```

Algorithm 3.1 describes the last step of the process: creating the slices. It sets up the camera according to the previously calculated bounding boxes of the input model on an orthogonal projection. After each rendering of that projection, a snapshot sized \( \sigma \) is created and saved onto the disk for further use. The camera then is moved along the \( z \) axis, and the process is repeated until the whole extension of the model has been rendered onto images. A visual representation of the variables mentioned and the slices made on an example model can be seen on Figure 1.

3.2 Polygonal model oversampling

Although we aim for a solution that accepts both polygonal models and point clouds, it is not possible to treat them as if they were unique. We focused ourselves on an approach that is easy to apply on point clouds, as they can be considered a simpler representation of a polygonal model.

We apply a simple oversampling operation that operates on a triangle level transforming a polygonal model into a perfect point cloud with a user-choice level of precision. Figure 3 shows an average polygon count model that describes this situation exactly. After discarding the polygons, we do not get a faithful representation of the original shapes, but after producing a point cloud through oversampling, the shape is exactly as the polygonal representation.

For each triangle \( \triangle abc \) on the input model, we check if the distance between any of its points is smaller than threshold \( t \), which represents the pixel size on the output image. If one meets this criteria, we are sure that any point inside this triangle is closer than \( t \) from an edge, otherwise we will apply the oversampling operation to it. Variables \( n₁, n₂ \) and \( n₃ \) represent the number of points to be created between two given points \( (x, y) \) in order to fit...
5 IMPLEMENTATION

Figure 3: Polygonal model, vertexes only, and generated point cloud

the threshold and are given by $\frac{\text{dist}(x,y)}{t}$.

Starting on point $a$, we start to create points along the edges $\overline{ab}$ and $\overline{ac}$. Each pair of points ($ab_i, ac_i$) created is then considered a new edge, and $n_i$ points are created along it.

When one of the edges is completed, we start creating points along $\overline{bc}$. The oversampled points created along this edge are interpolated with the points still being created on the uncompleted edge $\overline{ab}$ or $\overline{ac}$, thus filling the surface of the triangle.

The whole process applied to the input model creates a point cloud with precision defined by $t$. This guarantees us to have enough points to represent a closed surface without having unfilled pixels on the output image.

3.3 Information processing and encoding

Since all of our collision information will be written on collision maps as colors, we must assign each point on the point cloud a color representing its collision information. This will not replace the original color of that point in question. When writing these points on the output image on Algorithm 3.1, each pixel will represent a voxel sized $(t,t,\sigma)$ on object space. So the painted color on that pixel will represent all the points contained on that area. Algorithm 3.2 performs both the operation of height map information encoding, and obstacle detection.

The first operation is executed as follows: We calculate the difference between the current point $z$ coordinate and the models lower bounding box $z_{\text{min}}$, and apply the modulo operator with $\sigma$. This remainder $r$ represents the points $z$ coordinate on an interval $[0, \sigma]$.

To be used as a color value it must belong on the interval $[0, 1]$, so we calculate $\frac{r}{\sigma}$, deriving finally the red channel value. The simplified formula is given by $\text{red} \leftarrow \frac{\text{abs}(z - z_{\text{min}})}{\sigma}$.

As navigation on a real-world scenario is normally performed horizontally on the $xy$ plane, we classify an obstacle as a surface that is close to perpendicular to $xy$, parallel to $zy$. So our obstacle collision technique simply estimates how parallel to the $z$ axis a certain surface is. Figure 4 describes briefly this estimative operation executed on algorithm 3.2. Points lined up vertically on the same pixel most likely to belong to a vertical surface.

Using an auxiliary structure, the 3D array $\text{cube}[w][h][\sigma]$, after processing each point, we keep its color value on the position of the array representing the voxel on object space from

Algorithm 3.2 Points coloring and obstacle detection

for all points $p$ in $m$ do
  $s \leftarrow \text{floor}(\frac{\text{abs}(z - z_{\text{min}})}{\sigma})$
  $\text{red} \leftarrow \frac{\text{abs}(z - z_{\text{min}})}{\sigma}$
  $\text{red}_{\text{old}} \leftarrow \text{cube}[x_{\text{screen}}][y_{\text{screen}}][s]$
  if $\frac{\text{abs}(\text{red}_{\text{old}} - \text{red})}{\sigma} > \epsilon$ then
    $\text{cube}[x_{\text{screen}}][y_{\text{screen}}][s] \leftarrow 1$
    $p.\text{color}(\text{red}, 1, 1)$
    if $\text{red}_{\text{old}} > \text{red}$ then
      $p.z \leftarrow p.z + (\text{red}_{\text{old}} - \text{red}) * \sigma + \epsilon$
    end if
  else
    $\text{cube}[x_{\text{screen}}][y_{\text{screen}}][s] \leftarrow \text{red}$
    $p.\text{color}(\text{red}, 1, 0.1)$
  end if
end for
3 IMPLEMENTATION

Figure 4: Technique for surface orientation detection. Red points belong to a vertical structure, grey points to the floor.

where it came from. If there is already a stored value on this voxel, the difference between both red values is calculated, and transformed into an object-space distance $\frac{\text{abs(red}_{\text{old}} - \text{red}_{\text{new}})}{\sigma}$.

If this difference is bigger than a certain small percentage $\epsilon$ of the size $\sigma$ of the slice, we assume that the points are vertically aligned, belonging to a vertical surface. These points are marked on their blue channel with the value 1, and we slightly increase it’s $z$ coordinate in order to the point not be occluded on rendering time.

Some of the output slices from the pre-processing stage can be observed in Figure 5, an office environment, where the floor has been correctly assigned as green, and all the walls as white or light blue.

3.4 Collision detection

The developed representation provides us with enough information to perform quick collision detection on the navigation scenario given on subsection 1.1 where we consider point clouds as a viable input. While the aspects concerning the realism of the collision response have not been explored, precision on simple rigid bodies collision detection has been achieved.

We divide the task of collision detection into two steps: a first step, that we call Broad phase, where we verify the occurrence of collisions between any objects in the scene, and a second step called narrow phase, where we perform collision response.

3.4.1 Broad phase and collision detection

This task consists on identifying possible collisions between all objects on the scene. By representing the avatar that will be navigating on the environment by an Axis Aligned Bounding Box (AABB), we first calculate its size in pixels by calculating $\text{pix}_{x} \leftarrow \frac{\text{size}_{x}}{t}$ and $\text{pix}_{z} \leftarrow \frac{\text{size}_{z}}{t}$, where threshold $t$ was calculated as the pixel size. This will be the number of pixels checked for collision on each slice, around the center of the pawn. If any checked pixel is not black, we mark the object as colliding, and will be processed in a narrow phase.

The only images we will need to load into the memory at the same time in order to perform collision detection are from $\text{slice}_{0} \leftarrow \frac{\text{zpawn}_{\text{min}} + \text{zpawn}_{\text{max}} - \text{z}_{\text{min}}}{\sigma}$ to $\text{slice}_{n} \leftarrow \frac{\text{zpawn}_{\text{max}} + \text{zpawn}_{\text{min}} + \text{z}_{\text{min}}}{\sigma}$, the slices that contain information about the area where the pawn currently is.

New needed slices are onto memory until a user defined constant $n_{\text{slices}}$ is reached. New slices beyond this point, replace an already loaded slice that has the furthest $z$ value from the avatar’s own $z$ value, meaning it is not needed at this point of the execution.

3.4.2 Narrow phase and collision response

Our collision response algorithm has as it’s only objective to simply avoid objects from overlapping, and provide with a basic navigation experience on the given environment. This can be achieved with a simple extension to our broad-phase algorithm, by applying the concepts of collision response from height maps, and collision avoidance [20]. Instead of returning true when we find pixels that are not black, we gather information for collision response each time we find colored pixels.

Pixels with the blue channel set to 1 always represent an obstacle, except on applications where we want to enable the avatar to climb small obstacles, as the agents from Loscos et.al [20]. On these situations, we may ignore these pixels up until the height we want to be considered as climbable. As our avatar moves on
4 EXPERIMENTAL RESULTS

Figure 5: Three slices of an office environment, where walls and floor are clearly distinguished, as well as a subsection of a roof on the entrance.

fixed length steps, and each time it collides, we correct it to the place he was on the previous check, that we always assume as a valid position. We apply this \((x, y)\) correction each time an obstacle pixel is found until all the pixels representing the avatar’s bounding box are verified.

Height is defined exactly as it is on height maps. Multiplying the coded height information on the red channel by \(\sigma\) and adding the \(z\) base coordinate of the given slice, we have precise information about the given point’s height. Collision response can be made by setting the final height to the average height of the points on the base of the bounding box, or by the maximum value. Here also we check for surface height values from the first slice until the height we want to consider as climbable. The complexity of this operation is exactly \(O(\text{pix}_x \times \text{pix}_y \times s)\), but without adding any extra operation from the broad phase checking.

Seven models have been used for testing, each representing different scenarios and styles of modeling. None of them were tailored in a specific way to our application, as most of them were downloaded from free 3D models repositories on the web [2] [1]. A more detailed overview of them can be seen on Table 2.

Tests have been done for each model using two different image resolutions (700x700 and 350x350), performing obstacle detection (signaled by a + on Figure 6) and one without it (signaled by a - on Figures 6). This provides us with enough information about the key variables that make the difference on the efficiency of this stage. Resolution, obstacle detection, and polygonal oversampling.

Figure 6 shows the time taken on the whole preprocessing stage for each model and configuration. Polygonal oversampling is clearly the most heavy task on the pipeline, since the two tested point clouds had the faster performance. The increase on processing time with point clouds is linear to point complexity. This linear growth is expected since each point must be checked for coloring once, and also for common input processing such as input file reading and display list creation.

Regarding overall memory cost, we find that memory scales according to the size of the point cloud. Tests have shown that memory consumption was ordered according to the size of the produced point clouds. Table 3 shows the results of oversampling, and the size of the point clouds. The only extra memory needed

4 Experimental Results

4.1 Environment and settings

We have implemented the whole algorithm using OpenGL 2.1.2, C and the OpenGL Utility Toolkit (GLUT) to deal with user input and the base application loop managing. The platform used for testings is a computer with a Intel core 2 Duo CPU at 2 GHz with 2GB of RAM, a NVIDIA GeForce 9400 adapter, running Microsoft Windows Seven x86.
for our preprocessing stage is for allocating the auxiliary 3D array for obstacle detection.

During the application runtime, memory consumption varies according to the number of loaded slices onto the RAM, but by controlling $n_{slices}$ we can avoid this value from going over the memory we wish the application to consume. On a 700x700 resolution, the minimal value found was 81,92MB and the maximum 143,36MB , while on 350x350 values were between 61,44MB and 122,88MB. The complexity of the model has a much lighter impact here, being only noticeable on tasks that are unrelated to collision detection such as rendering and shading.

Results also show that our algorithm did not affect the rendering speed of the interactive application at all, environments where the frame rate was below the values considered minimum for interaction would be on this situation with any other collision detection algorithm applied to it. This shows that our technique is clearly not the bottleneck of the visualization cycle, one of the main concerns presented on subsection 1.1.

Results on collision detection have been ver-
Table 3: Polygonal oversampling results

<table>
<thead>
<tr>
<th>Model</th>
<th>Original</th>
<th>350x350</th>
<th>700x700</th>
</tr>
</thead>
<tbody>
<tr>
<td>Office</td>
<td>17.353 pts</td>
<td>3.088.193 pts</td>
<td>9.349.585 pts</td>
</tr>
<tr>
<td>Church</td>
<td>26.721 pts</td>
<td>2.246.549 pts</td>
<td>6.475.125 pts</td>
</tr>
<tr>
<td>Sibenik</td>
<td>47.658 pts</td>
<td>1.838.167 pts</td>
<td>5.199.093 pts</td>
</tr>
<tr>
<td>Columns</td>
<td>143.591 pts</td>
<td>1.448.383 pts</td>
<td>2.612.303 pts</td>
</tr>
<tr>
<td>Street</td>
<td>281.169 pts</td>
<td>3.606.225 pts</td>
<td>7.142.361 pts</td>
</tr>
</tbody>
</table>

ified through establishing a fixed route to navigate with the pawn where it goes through different situations and scenarios. Tests on Cathedral have showed us that reading from the disk on runtime has a bigger impact on efficiency than storing a higher number of images. When reading a high resolution image from disk, we notice a sudden drop on the frame-rate, and this is well noticed when the pawn falls from a higher structure. Increasing $n_{slices}$ to store enough slices to represent the ground floor and the platform on top of the steps, little to no difference was noticed on memory load, and the interaction was a lot smoother. On a low resolution though, reading from disk on runtime showed no impact on the performance.

Although we did not aim for high precision on collision response, our technique has presented precise results on different resolutions. Floor collision has been performed perfectly in all resolutions, as shown on Figure 7. Collisions with obstacles are more affected by resolution, since we rely on pixel finess to precisely know the position of a given surface. Tests on office and street have showed the same errors of small object interference or fake collisions due to diffuse information about object boundaries. These are more noticeable on the interactions with round objects on Street shown also on Figure 7, where we can notice the aliasing creating invisible square barriers around a perfectly round object.

Frame-rate was disturbed during the collision detection process on the R-tree aproach, while it remained steady at the 30 fps during the whole execution of our application. Also, the image-based technique has required much less memory to be executed, even with a high number of slices loaded onto memory. The biggest difference is in the pre-processing times. Our approach was executed 107 times faster than the BVH approach, and most importantly, this pre-processing stage must only be performed once for each configuration, since the representation is written to the hard disk and can be used on further interactions.

As stated on section 2 the research on point cloud collision detection is recent, and inexist-ent regarding image-based techniques. Our pioneer solution has presented excellent results, not only performing better than other works on point clouds published in the scientific community, but also being flexible enough to be applied on models from CAD, or combined with precise collision response techniques. Without adding any load to the visualization pipeline, our technique is not only scalable with input complexity, but also with hardware capabi-lities. Image-based collision detection can be performed with our representation on any computer that can render the input model at an acceptable frame-rate, without requiring anything meaningful from the CPU or GPUs.

Table 4 compares our technique with the work from Figueiredo et. al [11], which has has been tested on one of our experimental scenarios, the point cloud of the Entrance of the Batalha Monastery, on a machine with a slightly faster processing speed and RAM than the one used for our walkthroughs.
5 Conclusion and Future work

A new image-based environment representation for collision detection has been presented, using 2.5+D slices of an environment or building across the z axis. These images contain at each pixel, information about a certain voxel, representing it’s contents with colors. Height map information is stored on the red channel, and obstacles are indicated on the blue channel. This allows us to have a Broad phase collision detection stage that is performed with high efficiency and scalability, where the user can choose the precision according to the computing power at hand by simply adjusting the resolution of the produced images. Point clouds and polygonal models are ubiquitously processed, making our approach the top alternative for fast interaction with massive laser scan data. This work fills a gap in the area of collision detection, exploring a scenario that has been receiving more attention recently.

5.1 Future work

Implementing our broad-phase collision detection technique on the stencil buffer would greatly speed up this operation, since less testings would need to be made. Also, if we implemented the point coloring algorithm using a vertex shader, we wouldn’t have to perform the heavy memory allocation we need on the preprocessing stage, and also, calculations would be moved to the GPU, taking away some charge of the CPU.

Extending this representation to be used by ordinary objects such as the statue of David, or the Stanford Bunny, or even applying it to the avatars we’re using on the interaction, could present interesting results. Using non-uniform resolution images on environments where we do not have an uniform complexity, would also help us achieve more precision on our narrow phase, or on these presented situations.

Image comparison techniques can also be applied to the generated images in order to diminish the number of times we need to load a slice, and also the number of collision detection checks we must do.
References


REFERENCES


