# Self-Customized BSP Trees for Collision Detection*

Sigal Ar† Bernard Chazelle‡ Ayellet Tal§

**Abstract**

The ability to perform efficient collision detection is essential in virtual reality environments and their applications, such as walkthroughs. In this paper we re-explore a classical structure used for collision detection – the binary space partitioning tree. Unlike the common approach, which attributes equal likelihood to each possible query, we assume events that happened in the past are more likely to happen again in the future. This leads us to the definition of self-customized data structures. We report encouraging results obtained while experimenting with this concept in the context of self-customized BSP trees.

**Keywords:** Collision detection, binary space partitioning, self-customization.

## 1 Introduction

Virtual reality refers to the use of computer graphics to simulate physical worlds or to generate synthetic ones, where a user is to feel immersed in the environment to the extent that the user feels as if "objects" seen are really there. For example, "objects" should move according to force exerted by the user and they should not go through each other. To achieve this feeling of presence, one must address many issues, a key one among them is that of collision detection, which is fundamental in many other applications as well (e.g. [8, 4, 14]).

A large variety of data structures have been proposed for use in collision detection algorithms, with the goal of speeding up query processing and response time. They include, among others, simple grid hashing, k-d trees, BSP trees [20], the Dobkin-Kirkpatrick hierarchy[10], *R-tree* and its variants (e.g. [23, 25, 24]), the OBBtrees [13] and the Boxtrees [4]. In this paper, we return to explore the binary space partitioning – BSP – trees, a classical data structure used for collision detection.

Classical search trees, and BSP trees among them, have been extensively analyzed under worst-case and average-case models. The latter usually assume that keys are drawn randomly from a standard distribution (eg, uniform). Persuasive arguments have been made that such models are often realistic in practice. Three-dimensional geometry, however, paints a less pleasing picture. How does one justify a choice of distribution for a random 3D scene, a random navigation path in a walkthrough system, a random ray-shooting query in a BSP tree?

*The second author's work was supported in part by NSF Grant CCR-96-23768, ARO Grant DAAH04-96-1-0181, NSF Grant CCR-97-31535, and NEC Research Institute. The third author is a Milton & Lillian Edwards academic lecturer.

†Department of Electrical Engineering, Technion - Israel Institute of Technology

‡Department of Computer Science, Princeton University and NEC Research Institute

§Department of Electrical Engineering, Technion - Israel Institute of Technology

Even the simplest distributions (such as uniform among order types or input coordinates) are difficult to analyze and their practical relevance can be highly questionable.[1] People rarely navigate through a building by performing random walks and bouncing against walls aimlessly. Visibility or collision queries are typically dependent on previous ones, and we might be able to coax more predictive power from, say, a hidden Markov model [22] than we might from any closed-form distribution. But even there the geometry alone would be insufficient to specify the states and their transition probabilities. Exogenous factors can be just as important. Think how Mona Lisa's whereabouts might affect the walkthrough pattern in Le Louvre or how the showing of *Titanic* in a shopping mall might alter teenage crowd traffic. One might be able to use the geometry as conditioning data to produce relevant a posteriori information through some Bayesian model (eg, network beliefs [16, 21]). This would be one step towards injecting semantics into the model, which in the case of 3D environments is – we believe – often essential.

But even if we were to agree on a stationary client request distribution, how would we accommodate changes in client usage? More to the point, how would we use information about expected costs to optimize the data structure in real time (which is, after all, the whole purpose of the exercise)? Last but not least, how would we keep such a system simple and fast enough to be useful in practice?

In this paper we do not address the investigation of predictive models per se. Rather, we focus on the link between model acquisition and data structure optimization. We adopt a minimalist approach that avoids any a priori assumptions on client usage. This leads us to the concept of a *self-customized* data structure, which we briefly discuss below in full generality. For most of this paper, however, we use the BSP tree as a case-study to experiment with the idea of self-customization. The *binary space partitioning tree* [12], which generalizes kd-trees and oct-trees, is particular relevant to visibility computation, collision detection, and walkthrough systems [29]. It is also versatile enough to remain useful under various simplifying design assumptions. In other words, it is a good vehicle for experimenting with the self-customized concept in a relevant, flexible environment.

## Self-Customizing

Caching and prefetching are simple, powerful ideas in computer systems which can be found, under more complex guises, in a variety of data structures as well, such as buffer trees, splay trees, and other self-adjusting structures [2, 26]. All share the same basic traits, notably the assumption that events are more likely to happen in the future if they have already happened in the past. Self-customizing is premised on this temporal coherence principle alone. It is this minimalist approach that makes it particularly attractive. Just as spatial coherence is essential to fast rendering, temporal coherence has proven useful for collision and visibility, eg, [1, 8]. The main difference in our approach is that of scale. Our assumption is not that of "micro-coherence" (a fancy way of saying that functions should be piecewise smooth), but of coherence over longer periods of time. In our model, request distributions can be arbitrary and they can change over time arbitrarily. Changes, however, should be sufficiently spread apart to allow for effective statistical parameter estimation.

The idea behind a self-customized data structure is to hypothesize a probabilistic distribution of requests based on a log of recent client usage. That information is then used to reconfigure the data structure to improve performance. A self-customized data structure in

[1] Not always, of course. Classical point process analysis has been successful in astronomy and chemistry [5, 18].

action runs two concurrent processes:

- **Learning:** The system keeps a properly sampled log of the client's past requests and infers from it a probability distribution over the request domain. The distribution is periodically updated to reflect changing patterns in the log.
- **Reconfiguring:** At recurring intervals, the system updates the data structure – possibly rebuilding it from scratch – to optimize its expected request-answering complexity.

Both learning and reconfiguring activities must dovetail with client use, so as not to preempt access to the data structure at any time. Overhead should be kept minimal as a fraction of the overall request-answering costs. In other words, from the client's point of view, self-customizing should be a transparent, latency-free feature of the data structure.

BINARY SPACE PARTITIONING

Given a set $\mathcal{S}$ of disjoint polygons in $\mathbf{R}^3$, a BSP tree is a binary tree, where each node $v$ is associated with a plane $\pi_v$ and a closed convex polyhedron $C_v$. The root's polyhedron is a large box enclosing the scene $\mathcal{S}$. If $v$ is not a leaf, the plane $\pi_v$ cuts $C_v$ into the two convex regions associated with the children of $v$. Traditionally, leaves $v$ are characterized by the fact that no polygon of $\mathcal{S}$ should intersect the interior of $C_v$. We may relax this assumption, however, to save storage and add flexibility to the structure. The polygons of $\mathcal{S}$ are referred to as *scene polygons*. In practice, large ones should be triangulated and split into their constituent triangles.

Obviously, constructing a BSP tree is a highly nondeterministic process, which begs the question: what is a good construction for a given $\mathcal{S}$? BSP tree sizes can vary widely from linear to cubic. It is known that random constructions can limit the size to quadratic [20] and, it seems, much less in practice [6]. For this reason, our experiments focus on the other cost of a BSP tree, ie, the time for answering a ray-shooting query. To limit the scope of our investigation we restrict ourselves to auto-partitioning BSP trees: these are obtained by requiring each plane $\pi_v$ to contain a scene polygon.

Given a ray specified by a point $p$ and a direction $\ell$, we can find the first polygon of $\mathcal{S}$ that it hits by recursing on the following process. Assume that the ray is known to cross $C_v$. If $v$ is a leaf, then compute the answer by exhaustive examination of all the scene polygons that intersect $C_v$; otherwise, find whether the ray hits a scene polygon associated with node $v$. If it does, we have collision detection. (To find the front-most collision, recurse in the single child $w$ of $v$ whose polyhedron $C_w$ lies on the same side of $\pi_v$ as $p$.) If there is no collision in $v$, recurse in both its children.

## 2 Ray Distribution Learning

Given a training set $\mathcal{T}$ of $m$ rays, there are several choices for inferred distributions $\mathcal{D}$. Variants of standard methods [9, 11, 15, 22, 31] can be used. Our first choice derives from non-parametric maximum likelihood estimation: $\mathcal{D}$ is the discrete uniform distribution over the $m$ rays. It is simple – one might even say simplistic – but because distributions are to be used solely for collecting order statistics (not for sampling per se), it seems quite effective nevertheless.

For our other choices, we limit ourselves to translation-invariant distributions, which therefore can be fully defined over the sphere $S^2$. This is well justified in the context of, say, visibility

computations with a light source at infinity [3]. To generalize our experiments to arbitrary 5-dimensional rays is straightforward but more costly.

## 2.1 Multivariate-Gaussian

Assuming that the training points of $\mathcal{T}$ are represented by their $(x, y, z)$ coordinates, we form the covariance matrix $\Sigma = A^T A$, where $A$ is the $m \times 3$ matrix of points. Let $(u, v, w)$ be an orthonormal eigenbasis for $\Sigma$, and $\sigma_u, \sigma_v, \sigma_w$ be the corresponding eigenvalues. The inferred distribution can be sampled by the following simple procedure:

1. Draw random $U$ (resp. $V, W$) independently from the normal distribution with mean 0 and variance $\sigma_u$ (resp. $\sigma_v, \sigma_v$).
2. Choose as sample point of $S^2$ the point whose $(u, v, w)$-coordinates are

$$(U, V, W)/\sqrt{U^2 + V^2 + W^2}.$$

Suppose that the training set is uniformly distributed on $S^2$. Then, all eigenvalues are equal and the density function for $(U, V, W)$ is proportional to

$$e^{U^2/2\sigma_u + V^2/2\sigma_v + W^2/2\sigma_w},$$

which is spherically invariant. So, we end up sampling the unit sphere uniformly. Different eigenvalues produce a uniform sampling of the ellipsoid of inertia. From an implementation point of view, note that there is no need to diagonalize the covariance matrix $\Sigma$: we can sample $(x, y, z)$ from the unbiased normal distribution

$$f(x, y, z) = \frac{1}{\sqrt{(2\pi)^3 |\det \Sigma|}} e^{-\frac{1}{2}(x,y,z)\Sigma^{-1}(x,y,z)^T}.$$

and bring the sample point to $S^2$ by scaling. The strength of this method is its simplicity and robustness. Its weakness is to smooth out impulse distributions. In particular, it infers the uniform distribution from the one concentrated equally around three mutually orthogonal peaks. To avoid these problems we experiment with two variants of more sophisticated data analysis methods.

## 2.2 Cluster-Based Learning

We change our representation of $\mathcal{T}$ and now assume that points of $S^2$ are specified by their Euler coordinates $(\theta, \varphi)$, where $-\pi/2 \leq \theta \leq \pi/2$ is the latitude and $0 \leq \varphi < 2\pi$ is the longitude angle. Because $\varphi$ is mod $2\pi$, clusters might end up being split into two parts. There are several ways to go around this problem, such as making several copies of the Euler square: we choose the simpler approach of randomly rotating the coordinate system to decrease the likelihood of cluster splitting. Another, more serious, problem is distortion: to fix it, any further reference to area is to be understood in the spherical metric (with density $\cos\theta \, d\theta d\varphi$).

*k*-Means Clustering

Choose some parameter $k$ (the number of clusters), and pick $k$ random samples of size $\sqrt{n}$ in $\mathcal{T}$. Form the centroid (ie, mean point) of each sample; let $C$ denote the set of $k$ centroids. Iterate on the following process:

1. Compute the Voronoi diagram of $C$.

2. For each Voronoi cell, compute the centroid of the training points in it, and let $C'$ denote the set of $k$ centroids.

3. If $C$ is sufficiently different from $C'$, set $C \leftarrow C'$ and go back to 1.

The distribution is obtained by assigning a density to each Voronoi cell of the final $C$ by dividing the number of points in the cell by its (spherical) area and sampling accordingly.

The clustering is fairly sensitive to outliers and does not respond too well to overlapping clusters (although typically adding some amount of fuzzy clustering [15] can help). A more sophisticated approach is to integrate clustering with distribution inference through Gaussian mixture.

GAUSSIAN MIXTURE

Again, let $k$ denote the number of clusters. We postulate that the distribution $\mathcal{D}$ is a weighted sum of $k$ bivariate Gaussians. We estimate their means and covariances by the EM algorithm [9, 22]; it is a probabilistic version of $k$-means clustering. We seek a conditional probability function of the form

$$\text{prob}(x|\Pi) = \sum_{1 \le i \le k} w_i p(x|\mu_i, \Sigma_i),$$

where (i) $\Pi$ denotes the parameter vector $(\{w_i, \mu_i, \Sigma_i : 1 \le i \le k\})$, (ii) $w_i$ is the prior probability that the $i$th Gaussian was picked; (iii) $\mu_i$ is the mean of the Gaussian for the $i$-th class, (iv) $\Sigma_i$ is its covariance matrix, and (v) $p(x|\mu_i, \Sigma_i)$ denotes the probability density that $x$ is a point drawn from the Gaussian with mean $\mu_i$ and covariance matrix $\Sigma_i$. By definition,

$$p(x|\mu_i, \Sigma_i) = \frac{1}{2\pi\sqrt{|\det \Sigma_i|}} e^{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i)}.$$

The Expectation and Maximization stages of the EM algorithm are combined into one with the recurrence formula below. Initially, the parameter vector $\Pi_0$ is given the assignment corresponding to uniform weights ($w_i = 1/k$) and $\mu_i$ derived from the $i$-th class centroid derived from $k$-means clustering. All covariance matrices are set as $\sigma I$ (where $I$ is the $2 \times 2$ identity matrix) for small $\sigma$. After stage $t$, we compute the next assignment $\Pi_t$ for the parameter vector by using the recurrence formula:

$$\begin{cases} w_i^t &= \frac{1}{m} \sum_{j=1}^m p(i|x_j, \Pi_{t-1}) \\ \mu_i^t &= \frac{\sum_{j=1}^m x_j p(i|x_j, \Pi_{t-1})}{\sum_{j=1}^m p(i|x_j, \Pi_{t-1})} \\ \Sigma_i^t &= \frac{\sum_{j=1}^m p(i|x_j, \Pi_{t-1})(x_j - \mu_i^t)(x_j - \mu_i^t)^T}{\sum_{j=1}^m p(i|x_j, \Pi_{t-1})}. \end{cases}$$

The notation $p(i|x_j, \Pi_{t-1})$ refers to the posterior probability that the $i$-th Gaussian was chosen, given the observation $x_j$ and the belief that it was drawn from a Gaussian mixture with parameter vector $\Pi_{t-1}$; superscripts indicate stage index. The analogy with $k$-means clustering is easy to grasp. The mean $\mu_i$ of the $i$-th Gaussian is moved to the centroid of the data points, with each $x_j$ weighted in proportion to $p(i|x_j, \Pi_{t-1})$. By Bayes' rule, we have

$$
\begin{aligned}
p(i|x_j, \Pi_{t-1}) &= \frac{p(i|\Pi_{t-1})p(x_j|i,\Pi_{t-1})}{p(x_j|\Pi_{t-1})} = \frac{p(i|\Pi_{t-1})p(x_j|i,\Pi_{t-1})}{\sum_\ell p(\ell|\Pi_{t-1})p(x_j|\ell,\Pi_{t-1})} \\
&= \frac{w_i^{t-1}\, e^{-\frac{1}{2}(x_j-\mu_i^{t-1})^T(\Sigma_i^{t-1})^{-1}(x_j-\mu_i^{t-1})}}{\sum_\ell w_\ell^{t-1}\, e^{-\frac{1}{2}(x_j-\mu_\ell^{t-1})^T(\Sigma_\ell^{t-1})^{-1}(x_j-\mu_\ell^{t-1})}}.
\end{aligned}
$$

# 3 Tree Configuration

Dynamic updating of BSP trees has been extensively studied [1, 7, 27, 30], and there is no need here for detailed discussion of the primitive geometric operations involved. Given a node $v$ of a BSP tree for $\mathcal{S}$, let $P_v$ be the convex polygon formed by the intersection of the cutting plane $\pi_v$ with the convex polyhedron $C_v$. The traversal cost of a directed line in a BSP tree is the number of nodes that are "opened" before the first collision (with a scene polygon) is found. This is at most proportional to the height of the tree added to the number of polygons $P_v$ that the line intersects. We ignore the first term since it is the same for all queries and since the second one is typically dominant. (Because rays do not extend past scene polygons, this number overcounts the actual cost: this is not a problem as long as the overcounting is roughly the same for most directed lines.)

Our experimentation reveals that the following measures have a direct influence on the traversal cost, and therefore on the "goodness" of a plane as a candidate cutting plane. One key parameter is the projected surface area of a scene polygon along the viewing direction. More specifically,

- The angles at which the rays hit the plane. For a given ray and plane, the closer this angle is to 90°, the more likely the ray is to hit a scene polygon embedded in that plane.
- The total surface area of the scene polygons that are embedded in the candidate cutting plane, relative to the total scene area. The more surface area the plane "covers", the more likely it is that corresponding scene polygons are hit by rays aimed at the scene.
- The position of the plane relative to the source point(s) of the rays and relative to the other planes of the scene. Rays are blocked from planes that are behind other planes, relative to their source point.
- The surface area of the $P_v$'s (the intersection of cutting plane with the convex polyhedra associated with the nodes). Rays aimed at the general direction of the node are more likely to hit a polygon with a bigger surface area.

Let $\omega(\ell)$ be the density measure of the ray distribution $\mathcal{D}$. We solve the corresponding optimization problem by following a randomized greedy strategy. We build the BSP tree incrementally by inserting planes one at a time. At any time during the construction, we keep a list $L$ of the scene planes (ie, the planes coplanar with polygons of $\mathcal{S}$) that are potential cutting planes, and we maintain the *score* of each scene plane $\pi$, defined as

$$
\mathit{score}(\pi) = \int_\ell \mathit{weight}(\mathcal{S}, \pi, \ell)\, \omega(\ell)\, d\ell \tag{1}
$$

for some *weight* function measuring the desirability of a cutting plane.

The scoring can be done in one of two modes. In the *prescoring mode*, each potential plane is given a score once, before we begin building the BSP tree. These scores are used throughout the tree construction. In the *incremental scoring mode*, we recalculate a plane's score each time it is a candidate cutting plane.

### 3.1 Prescoring Mode

In the prescoring mode, we start by calculating $score(\pi)$ for each plane $\pi$ defined by a scene polygon. Let $(\vec{\pi}, \ell)$ denote the angle between $\ell$ and and the normal to plane $\pi$. Let area$(\mathcal{S})$ be the total area of all scene polygons, and let area$(\mathcal{S} \cap \pi)$ be the area of scene polygons on the plane $\pi$. We experimented with several weight functions. The one that proved best in practice was

$$weight(\mathcal{S}, \pi, \ell) = (1 - \cos(\vec{\pi}, \ell))^2 \cdot \frac{\text{area}(\mathcal{S} \cap \pi)}{\text{area}(\mathcal{S})} \tag{2}$$

This corroborates the intuitive belief that favoring scene polygon hits high up in the tree limits the branching factor in answering a query and therefore lowers costs. Let $L_v$ denote the set of scene planes crossing $C_v$. To select the cutting plane $\pi_v$ for a current leaf $v$ that we wish to split, we choose among $L_v$ as follows:

- Sort the planes $\pi \in L_v$ by $score(\pi)$.
- Randomly choose one plane among the highest-scoring planes.

Note that the above score function aims to hit near the root of the BSP tree. Other score functions can be used. For instance, if one wants to optimize for rays aimed at the general direction of the scene, but missing its polygons, the weight function can be altered accordingly without difficulty.

### 3.2 Incremental Mode

A more complex measure of costs includes the actual $P_v$'s and requires on-line weight updates. By translation invariance, the probability that a directed line of direction $\ell$ intersects the polygon $P_v$ is proportional to area $(P_v)|\cos(\vec{\pi}_v, \ell)|$, so the "cost" of $P_v$ is represented by

$$\int \text{area}\,(P_v)|\cos(\vec{\pi}_v, \ell)|\, d\omega(\ell). \tag{3}$$

Note that in the case of a uniform distribution we find the cost is proportional to the sum of the areas, which agrees with previous observations [3, 4]. At any time during the construction, we keep a list $L_v$ of the scene planes crossing $C_v$, for each current leaf $v$. We redefine the weight function to be recalculated at each selection step, and the score of each scene plane $\pi$ is now defined as

$$score(\pi) = \sum_{v\,:\,\pi \in L_v} \int \text{area}\,(C_v \cap \pi)|\cos(\vec{\pi}, \ell)|\, d\omega(\ell).$$

Our plane selection strategy is to draw a random scene plane (among those remaining to be inserted) from the distribution induced by the score function. This time, we seek to minimize the score function in order to limit the area of the non-scene polygons (weighted by the ray distribution). We define the probability of picking a plane $\pi_0$ at stage $k$ to be

$$\frac{1}{score(\pi_0)\sum_{\pi} 1/score(\pi)}.$$

For technical reasons it is preferable to threshold the tail of the distribution and renormalize it, so that each probability remains within $[0,1] \cap [\varepsilon/(n-k), 1/\varepsilon(n-k)]$, for some small constant $\varepsilon > 0$.

The randomization in our strategy serves two purposes. One is to avoid falling into narrow local minima. The other is to avoid a storage blowup.

# 4 Experimental Results

Variants of the algorithms involved in our classification and BSP tree constructions were implemented in C on a Silicon Graphics workstation. To guide our search for good weight functions, and to visualize the results, we used the geometric animation system GASP [28]. In Figure 1 we show some common objects that were used as our scenes.

To simulate the learning process, we produced training data, assuming a multivariate normal distribution. Then, to benchmark the performance of a given BSP tree, we produced collision queries using distribution parameters similar to those used for the training data.

We compared BSP trees produced using our scoring mechanism to some highly optimized BSP trees that have been discussed in the literature. In particular, we compared our customized BSP trees in the prescoring mode to what we shall call *the standard* BSP trees of the Graphics Gems [19], where the main concern is to minimize the number of intersections of cutting planes. For the sake of meaningful comparisons, apart from the choice of the cutting planes, we applied to the standard BSP the same optimizations we used on our customized BSP code.

Table 1 below presents the results of our experiments. Next to each object, we indicate the number of faces comprising it. For each scene, we ran the Standard BSP building code and our new Customized BSP building code. They were compared under several configurations. We varied the number of rays and the number of clusters in which they fall. (Both training data and queries were allowed to vary, not always in the same way.) To compare the performance, we counted the number of BSP tree nodes opened (for all the rays) in each case - the Customized BSP vs. the Standard BSP .

Our experiments indicate that self-customized BSP trees can offer stunning speedup factors. For instance, when one cluster of 1000 rays was aimed at the back end of the transporter, the Standard BSP opened 135,663 BSP tree nodes, while our Customized BSP opened only 1000 nodes. As another example, when four clusters of 1500 rays (total) were aimed at the side of the wheel, some hitting the spokes and some hitting the tire, the Standard BSP opened 30,085 nodes, while the Customized BSP opened 8771. Our advantage comes from the training data, that suggests picking the first BSP cutting plane as the one that the rays are actually expected to hit first.

Two related factors we did not attempt to optimize in the actual experimentation with customized BSP trees are the tree size and the tree construction time. The customized BSP trees have, on average, twice as many nodes as the standard BSP trees. Even so, when the customized BSP trees are optimized to favor hitting rays, they perform much better. As for the preprocessing, the construction time for the customized trees is much smaller than that needed for the corresponding standard BSP trees. This is because calculating intersections takes much longer than calculating our scores, regardless of relative tree sizes.

| A Sphere (556 faces) | | | |
|---|---|---|---|
| No. of Rays | No. of Clusters | Standard BSP | Self-Customized BSP |
| 100 | 1 | 15,055 | 3,008 |
| 200 | 4 | 38,807 | 5,293 |
| 1,000 | 4 | 189,804 | 24,073 |
| A Wheel (972 faces) | | | |
| No. of Rays | No. of Clusters | Standard BSP | Self-Customized BSP |
| 100 | 1 | 2,617 | 1,803 |
| 150 | 2 | 3,637 | 2,870 |
| 300 | 1 | 7,200 | 300 |
| 1,500 | 4 | 30,085 | 8,771 |
| A Transporter (3,952 faces) | | | |
| No. of Rays | No. of Clusters | Standard BSP | Self-Customized BSP |
| 100 | 2 | 16,156 | 1,251 |
| 1,000 | 1 | 135,663 | 1,000 |
| 1,000 | 3 | 161,206 | 11,114 |
| The St. Pauls Cathedral (3,865 faces) | | | |
| No. of Rays | No. of Clusters | Standard BSP | Self-Customized BSP |
| 100 | 1 | 1,816 | 580 |
| 300 | 3 | 7,598 | 2,061 |
| 800 | 2 | 18,584 | 8,362 |
| A Robot (2,294 faces) | | | |
| No. of Rays | No. of Clusters | Standard BSP | Self-Customized BSP |
| 750 | 2 | 19,879 | 14,407 |
| 1,000 | 1 | 7,232 | 1,483 |
| 1,000 | 3 | 30,876 | 14,293 |
| A Shuttle (558 faces) | | | |
| No. of Rays | No. of Clusters | Standard BSP | Self-Customized BSP |
| 100 | 1 | 4,906 | 1,944 |
| 750 | 2 | 44,235 | 28,680 |
| 1,200 | 3 | 51,667 | 26,858 |
| A Human Head (1,850 faces) | | | |
| No. of Rays | No. of Clusters | Standard BSP | Self-Customized BSP |
| 100 | 1 | 6,348 | 1,318 |
| 900 | 3 | 53,890 | 14,434 |
| 1,200 | 5 | 70,024 | 18,902 |

Table 1: Comparison of the number of nodes opened

# 5 Conclusions

We introduced the concept of *self-customized* data structures, and investigated it in the case of BSP trees for collision detection. Self-customizing requires two steps: (i) learning from a sample of client requests to infer a distribution over the request domain; (ii) (re)configuring the data structure to optimize its expected request answering costs.

We discussed various ways to infer the distribution parameters. We also showed how to devise, from a distribution and from some information about the scene, a scoring mechanism for scene planes.

To benchmark the performance of self-customized BSP trees on collision detection queries, we compared the query answering costs of the customized BSP trees to those of some public domain BSP trees.

On the basis of our experimental results, it is clear that self-customizing can greatly improve performance of query response time. Moreover, if the self-customized data structure is optimized with respect to the inferred distribution, then its size becomes insignificant to query response time.

We believe self-customization is worthy of further investigation. Potential other domains can include point location and range searching. In addition, we intend to investigate the combination of deferred data structuring [17] with self-customization.

# References


[1] Agarwal, P.K., Guibas, L.J., Murali, T.M., Vitter, J.S. *Cylindrical static and kinetic binary space partitions*, Proc. 13th Annu. Symp. Comput. Geom. (1997), 39-48.

[2] Arge, L. *The buffer tree: a new technique for optimal I/O-algorithms*, in "Algorithms and Data Structures," eds. Akl, S.G., Dehne, F., Sack J.-R., Santoro, N., Springer (1995), 334–345.

[3] Aronov, B., Fortune, S. *Average-case ray shooting and minimum weight triangulations*, Proc. 13th Annu. ACM Symp. Comput. Geom. (1997), 203-211.

[4] Barequet, G., Chazelle, B., Guibas, L.J., Mitchell, J., Tal, A. *BOXTREE: a hierarchical representation for surfaces in 3D*, Graphics Forum, 15 (1996), C–387–396.

[5] Bartlett, M.S. *The Statistical Analysis of Spatial Pattern*, Chapman and Hall, London, 1975.

[6] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*, Springer, 1997.

[7] Chrysanthou, Y., and Slater, M., *Computing dynamic changes to BSP trees*, Computer Graphics Forum (EUROGRAPHICS '92 Proceedings), 11 (1992), 321–332.

[8] Cohen, J., Lin, M., Manocha, D., Ponamgi, K. *I-COLLIDE: an interactive and exact collision detection system for large-scaled environments*, Proc. ACM Int. 3D Graphics Conf. (1995), 189–196.

[9] Dempster, A., Laird, N., Rubin, D. *Maximum likelihood from incomplete data via the EM algorithm*, J. Royal Statistical Society, B 39 (1977), 1–38.

[10] D.P. Dobkin and D.G. Kirkpatrick, *Fast detection of polyhedral intersection*, Theoret. Comput. Sci., **27** pp. 241–253 (1983).

[11] Duda, R.M., Hart, P.E. *Pattern Classification and Scene Analysis*, Wiley, 1973.

[12] Fuchs, H., Kedem, Z.M., Naylor, B. *On visible surface generation by a priori tree structures*, Proc. SIGGRAPH '80, Comput. Graph., 14 (1980), 124–133.

[13] Gottschalk S., Lin, M.C., and Manocha, D. *OBBTree: A Hierarchical Structure for Rapid Interference Detection*, Proc. SIGGRAPH '96, 171–180.

[14] M. Held, J.T. Klosowski, and J.S.B. Mitchell, *Evaluation of collision detection methods for virtual reality fly-throughs*, Proc. 7th Canadian Conf. Computational Geometry, pp. 205–210 (1995).

[15] Jain, A.K., Dubes, R.C. *Algorithms for Clustering Data*, Prentice Hall, 1988.

[16] Jensen, F. *An Introduction to Bayesian Networks*, Springer, 1996.

[17] Karp R.M., Motwani R. and Raghavan P. *Deferred Data structuring*, SIAM Journal on Computing, 17 (1988), pp. 883-902

[18] Okabe, A., Boots, B. and Sugihara, K. *Spatial Tesselations*, Wiley, New York, 1992.

[19] Paeth A.W. *Graphics Gems V*, Academic Press, 1995.

[20] Paterson, M.S., Yao, F.F. *Efficient binary space partitions for hidden-surface removal and solid modeling*, Disc. Comput. Geom., 5 (1990), 485–503.

[21] Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, 1988.

[22] Rabiner, L., Juang, B.H. *Fundamentals of Speech Recognition*, Prentice-Hall Signal Processing Series, 1993.

[23] N. Roussopoulos and D. Leifker, *Direct spatial search on pictorial databases using packed R-trees*, Proc. ACM SIGACT-SIGMOD Conf. Principles Database Systems, pp. 17–31 (1985).

[24] H. Samet, *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*, Addison-Wesley, Redding, Mass., 1989.

[25] T. Sellis, N. Roussopoulos, and C. Faloutsos, *The $R^+$-tree: A dynamic index for multidimensional objects*, Proc. 13th VLDB Conf., pp. 507–518 (1987).

[26] Sleator, D.S. and Tarjan, R.E. *Self-adjusting heaps*, SIAM J. Comput. 15 (1986).

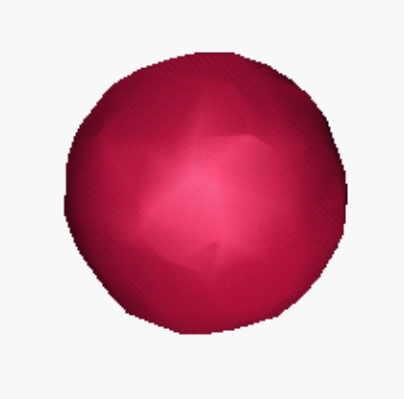
[27] Sung, K., Shirley, P. *Ray tracing with the BSP tree*, ed. David Kirk, Graphics Gems III, Academic Press Inc. (1992), 271–274.

[28] Tal, A. and Dobkin, D. P. *Visualization of geometric algorithms*, IEEE Trans. on Visualization and Computer Graphics, 1 (1995), 194–204.

[29] Teller, S.J., Sequin, C.H. *Visibility preprocessing for interactive walkthroughs*, Proc. SIGGRAPH '91, Comput. Graph., 25 (1991), 61–69.

[30] Torres, E. *Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes*, EUROGRAPHICS '90, Eurographics Association (1990), 507–518.

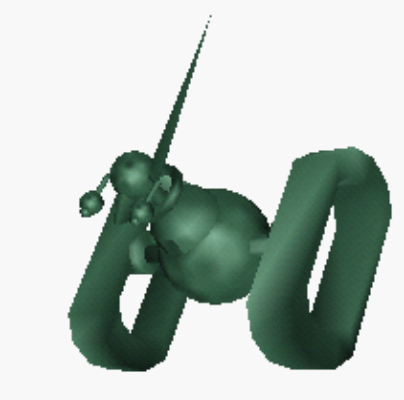
[31] Upton G., Fingleton B., *Spatial Data Analysis by Examples*, Wiley 1985.



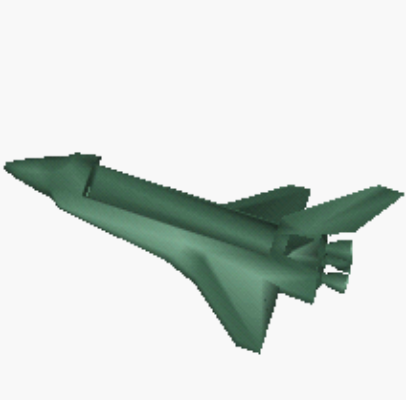
(a) A Sphere

(b) A Wheel

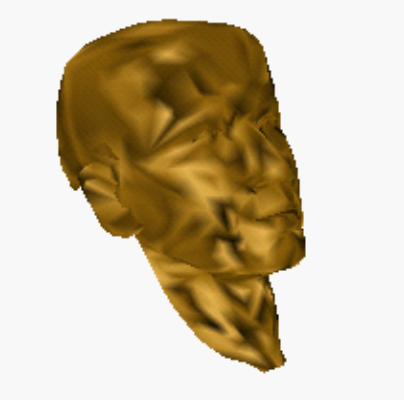
(c) A Transporter



(d) A Robot



(e) A Shuttle



(f) A Human Head

(g) The St. Paul's Cathedral

Figure 1: The objects used