INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Automatic Cell-and-portal Decomposition

Sylvain Lefebvre — Samuel Hornus

N° ????

Juillet 2003

THÈME 3

ISSN 0249-6399

Automatic Cell-and-portal Decomposition

Sylvain Lefebvre , Samuel Hornus

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projets Evasion et Artis

Rapport de recherche n° ???? — Juillet 2003 — 20 pages

Abstract: We present a method to automatically compute a decomposition of a polygonal scene into a simple cell-and-portal graph. The resulting cell-and-portal graph satisfies the following user-defined constraints: an upper bound on the rendering cost of each cell, and lower or upper bounds on the size of each cell. This is useful to achieve real-time rendering of large indoor models, and is especially suited to architectural walk-throughs and game engines. Our method relies on a binary space-subdivision preprocessing step, then on a portal grouping algorithm that selects or rejects portals generated by the subdivision. Finally the cell-and-portal graph (CPG) is built and post-processed to satisfy the constraints on the cells. We also propose a metrics for measuring the quality of portals, which is used to guide the post-processing. Furthermore, our simplification algorithm can be used on any CPG in order to reduce its complexity according to a user threshold. We present both a general algorithm and a complete implementation with practical details. Results show that portals created by our method have good geometrical properties (e.g. they often lie on doors and windows). The generated decomposition can be used for online occlusion culling.

Key-words: Subdivision, cells and portals, conservative visibility, architectural walk-through

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Décomposition automatique de scène en cellules et portails

Résumé : Nous présentons une méthode de décomposition automatique d'une scène polygonale en un graphe de cellules et portails (CPG). Le graphe résultant satisfait deux contraintes définies par l'utilisateur : une borne sur le coût de rendu de chaque cellule, et des bornes inf. et sup. sur la taille de chaque cellule. Cette décomposition permet le rendu en temps-réel de grandes scènes d'intérieur, et est plus particulièrement adaptée aux ballades virtuelles dans les scènes architecturales, telles que dans les jeux vidéos. Notre méthode est basée sur une subdivision binaire de l'espace, puis sur un algorithme sélectionnant et regroupant certains portails générés par la subdivision. Enfin, le CPG est construit et traité pour satisfaire les contraintes données par l'utilisateur. Nous proposons également une métrique pour mesurer la qualité d'un portail, que nous utilisons pour le traitement final du graphe. Notre algorithme de simplification peut aussi être utilisé sur n'importe quel CPG pour réduire sa complexité. Nous présentons la méthode générale et quelques détails pratiques d'implémentation. Les résultats montrent que les portails générés ont de bonnes propriétés géométriques (ils sont la plupart du temps placés sur les portes et les fenêtres).

Mots-clés : Subdivision, cellules et portails, visibilité conservative, ballade virtuelle

Automatic cell-and-portal decomposition

Sylvain Lefebvre
EVASION[†], lab GRAVIR/IMAG
INRIA Rhône-Alpes, France
Sylvain.Lefebvre@imag.fr
, Samuel Hornus
ARTIS[†], lab GRAVIR/IMAG
INRIA Rhône-Alpes, France
Samuel.Hornus@imag.fr

August 1, 2003

We present a method to automatically compute a decomposition of a polygonal scene into a simple cell-and-portal graph. The resulting cell-and-portal graph satisfies the following user-defined constraints: an upper bound on the rendering cost of each cell, and lower or upper bounds on the size of each cell. This is useful to achieve real-time rendering of large indoor models, and is especially suited to architectural walk-throughs and game engines. Our method relies on a binary space-subdivision preprocessing step, then on a portal grouping algorithm that selects or rejects portals generated by the subdivision. Finally the cell-and-portal graph (CPG) is built and post-processed to satisfy the constraints on the cells. We also propose a metrics for measuring the quality of portals, which is used to guide the post-processing. Furthermore, our simplification algorithm can be used on any CPG in order to reduce its complexity according to a user threshold. We present both a general algorithm and a complete implementation with practical details. Results show that portals created by our method have good geometrical properties (e.g. they often lie on doors and windows). The generated decomposition can be used for online occlusion culling.

1 Introduction

Subdivision of a scene is a widely used method in computer graphics. It helps with organizing information about the scene and therefore reduces the computational complexity of further calculations. It is specifically useful for the efficient computation of various visibility informations. A typical application is the display of large scenes at interactive rates. For example, the subdivision of the scene into smaller components, together with visibility relationships, is used to avoid the drawing of the whole model [14]. In radiosity computations, one needs to compute the set of objects visible from

[†] ARTIS and EVASION are joint research projects of CNRS, INPG, INRIA, UJF.

a certain region of space, such as a light source [4, 10]. It is also a convenient way to pre-process visibility queries used in global illumination [9, 15], allowing the computation of energy exchanges only where appropriate.

To solve particular visibility queries, subdivision structures like binary space partition trees (BSP trees), hierarchical bounding boxes, and cells and portals, are often used. A BSP is a scheme for recursively dividing a scene by *cutting* planes. One generally want small or well balanced BSP trees, but such optimal trees are difficult to compute [13]. Therefore heuristics are used to find good cutting planes among the planes supporting the scene's polygons. Hierarchical bounding boxes are widely used for view frustum culling of complex objects. For walk-through applications, cell-and-portal graphs (CPGs) provide connectivity and visibility information. Cells are polyhedra, whose facets are polygons from the model, or portals. A portal is a "transparent" polygon [1] connecting two adjacent cells, explicitly marking the connectivity and visibility relations between cells. A BSP tree may still be used, e.g. for collision detection, or front-to-back ordering of the polygons of simple objects.

BSP trees are widely used in the game industry to perform off-line conservative visibility computation (computation of potentially visible sets: PVS). Tools like `qbsp3`, `qvis3` [2], `zhlt` [3], are freely available and used for many games. However, as the size of video-games levels grows larger, the BSP/PVS computation becomes too slow for production, due to the growing size of the BSP tree, and the high computational complexity of PVS calculation. This explains the important shift toward the use of CPG and online occlusion culling in recent games (e.g. Doom III/id software). In a good CPG, cells are larger and the portals fewer, enabling the use of more efficent rendering methods (e.g. "portal rendering", see below). While good BSP tree construction is simple to implement, good CPGs are difficult to obtain automatically. They can be constructed by hand during the modeling process, using tools found in modeling applications, but it remains a tedious task for the user. Also, many existing models do not include any partitioning information, and we would like to construct such information. As a CPG is a simple graph embedded in a three-dimensional scene, it lacks a strong structure (as opposed to BSP trees). This makes the definition of a *good CPG* hard to state. Very specialized algorithms have been developed using common knowledge rules on the size of rooms to construct cells from a floor plan [9]. They are not general and cannot create portals with arbitrary orientations. We are not aware of the existence of publicly available tools to generate CPGs for architectural scenes. Such tool would be very convenient to speed up both PVS calculations (for off-line preprocessing) and online visibility determination.

1.1 Technical motivation for the proposed method

We define a *watertight* scene as a 3d model whose *interior* is well defined. Think of the interior of a 3d model as the space that would be occupied by real material (concrete, wood, etc...). Polygonal surfaces in a watertight scene have no holes: such holes would destroy the separability between the interior and the exterior of the scene. Let a watertight scene be decomposed in a BSP tree. Each leaf of the tree is a closed convex polyhedron whose facets either belong to an initial polygon of the

[1] they help partitioning the model, but should not be drawn
[2] http://www.planetquake.com/lfi re/Level_Editing/Tutorial/tutorial1.html
[3] http://collective.valve-erc.com/

scene or are *empty/transparent* facets adjacent to another leaf. Therefore, we can see this BSP tree as a cell-and-portal decomposition. However, such a decomposition is not adapted for interactive walk-through in large scenes; this is explained in the following.

Given some cell-and-portal decomposition, there are two main methods to perform occlusion culling, one is online (portal rendering), the other off-line (PVS). Given a viewpoint in cell C, portal rendering proceeds as follows:

1. draw the polygons of cell C.
2. for each portal p of cell C,
   if p is visible then
   recursively call the procedure for the
   back cell of p.

Various methods are used for the [*portal is visible*] oracle [2, 8, 14]. For portal rendering, using the BSP as a cell-and-portal decomposition is not adapted: many cells have very few polygons, and the number of portals can be dramatically high: too much time is spent determining if portals are visible. If the cells are large enough and there are few portals (which is *not* the case with BSP), portal rendering benefits from the high bandwidth/fill-rate of recent graphic hardware, because we can use bandwidth optimization methods such as vertex arrays with fast memory access or simple display lists: In this way, the geometry of each cell is quickly sent to the graphic board as we traverse the graph. Besides, recent graphic boards [12] include an occlusion query functionality that makes portal rendering easy to implement : portal visibility queries can be entirely done in hardware.

For off-line rendering, each leaf of the BSP tree is given a *potentially visible set* (PVS) of leaves. The PVS encodes for each leaf L the set of leaves that are potentially visible from L, that is, visible from at least one point inside L. Each leaf contains its PVS. If the viewpoint lies in leaf L, rendering the scene is therefore just a matter of drawing the polygons of all the precomputed potentially visible leaves of leaf L. This method was introduced by Airey *et al.* [1, 2] and Teller and Séquin [16].

We can have up to $O(n^2)$ leaves [13] and $O(n)$ visible polygons per leaf (although usually much less), where n is the number of polygons in the scene. We would like to benefit from the large memory of recent graphic boards and avoid bandwidth problems; Storing the geometry in the graphic hardware is a solution, but appears difficult to handle if we use PVS for rendering, because there would be too many (quite large) display lists. Portal rendering however, could be used, because each polygon of the scene would be stored only once in graphic memory.

Interesting works have been done on PVS compression. Van de Panne and Stewart [17] obtain very good (lossy or not) commpression ratios. Our goal however is also to reduce preprocessing time, so we are looking for alternatives to PVS computations.

To overcome the problems appearing with both online and off-line occlusion culling for today's indoor complex scenes – some levels in Unreal Tournament 2003 are made of more than 150,000 polygons – we would like to generate cell-and-portal graphs with bigger cells. There is a need for a cell-and-portal decomposition method, capable of handling general 3D architectural scenes and giving some control on cells' complexity.

We propose a method to automatically build a *simple* cell-and-portal decomposition of an arbitrary architectural scene. The method gives some control on the final CPG to the user, permits *arbitrary* orientation for the portals, and produces cells suitable for *on-line* occlusion culling ("por-

tal rendering"). We present a heuristic to compare the quality of portals. The heuristic is devised from a short analyse of what a good CPG should be for fast rendering.

Overview of the paper. In section 2, we present related work on cell-and-portal graphs. Section 2 outlines the method we propose for creating a CPG from the *raw* scene. Section 3 presents our CPG simplification algorithm and the metrics we use to compare the quality of the portals. Section 3.0.2 discusses our implementation of the method on polygonal input scenes and presents some of our results. We conclude and give some possible future work in section 4.

2 Previous work

Jones introduces the cell-and-portal decomposition of the space surrounding an object in 1971 [7], as a solution to the hidden-line problem. He manually decomposes the space into convex cells so that the faces of the object lie on faces of the cells. The rendering of the object then proceeds roughly as described in the introduction.

Airey *et al.* and Teller *et al.* [2, 16] introduce the cell-and-portal decomposition as a way to efficiently render architectural scenes. The method is then further analyzed and improved by Teller for 2D, 3D axial and general 3D scenes [14, 16]. He presents methods and heuristics to compute a BSP of a scene, and, for each leaf (or cell) of the BSP tree, compute the set of potentially visible cells (so called cell-to-cell visibility information).

For general (i.e. non axis-aligned) 3D indoor scenes, Teller uses a BSP tree with some heuristics to compute a decomposition of the scene into cells and portals. This is sufficient and even well suited (because of the convexity of the cells) for the PVS computation he proposes, but the scene is, in general, way too subdivided for on-line "portal rendering" without PVS. Fuchs *et al.* [5] did the initial work on the use of BSP-trees and on heuristics to construct them.

Luebke and Georges [8] show a method for efficient on-line occlusion culling for eye-to-cell visibility with a CPG, using a 2D image-space portal bounding box to cull the geometry of a cell. The method, known as "portal rendering", is used in the game industry because of its efficiency, but, as far as we know, the CPG is still generated manually.

In 1997, Meneveaux et al. [9] propose a new method to decompose a 3D scene into a cell-and-portal graph. Their approach projects the vertical walls on the horizontal plane and clusters the resulting segments in the dual space. They use special rules based on architectural knowledge to build cells one after the other, and then compute the portals. The method is fast, but cannot be applied to general scenes since the decomposition really is two dimensional.

James *et al.* [6] give a method to simplify a BSP tree to speed up the rendering of architectural scenes. Their method do not explicitely produced a CPG, but the idea of merging BSP cells to obtain a good decomposition of the scene is there.

The method we propose automatically generates a simple CPG, using an initial BSP of the scene, as illustrated in figure 1. It is interesting to note that in contrast to previous approaches, our method *starts* by looking for "good portals" and *then* builds the cells using connected components of a graph of smaller cells.

Given an architectural scene, we first add separators ("transparent" polygons) in it with good geometrical properties. Intuitively, newly added separators must hermetically close some path in the

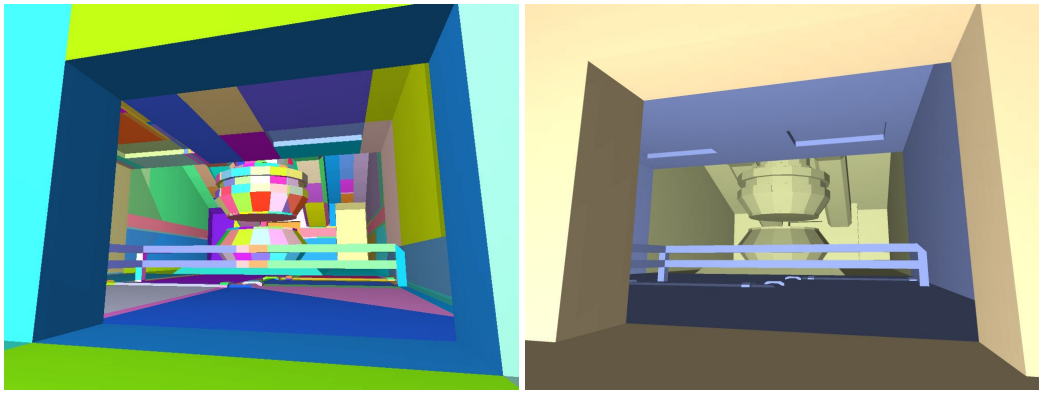

Figure 1: **Left:** BSP of a scene. **Right:** our cell-and-portal decomposition usable for walk-through applications. Three cells are visible.

scene. This is an interesting property for on-line visibility determination as it ensures that nothing can be visible behind a separator if it is hidden. These separators (portals) define a cell-and-portal graph: The cells correspond to the volumes enclosed between the separators and the polygons of the scene. We create the separators from the boundaries of some initial subdivision of the scene. There exist many different methods to subdivide a scene, but for our purposes the scene polygons must match the facets of the initial subdivision. Therefore we use a standard BSP tree (see section 3.0.2).

The cells we create are generally *non convex* . This does not prevent the use of portal rendering methods or PVS computations nor some other optimizations like occlusion culling [3], but the results can be less accurate (i.e. the superset of visible objects can be larger). However, with our method, complex geometry such as stairs will generally be captured as a single cell, which is not possible if we require cells to be convex.

Our method is decomposed in two algorithms: the *grouping algorithm* first creates a CPG from an initial BSP tree. All the portals of this CPG are separators with good geometrical properties as defined above. The *simplification algorithm* then simplifies the CPG by only keeping relevant portals. The grouping algorithm is decomposed in four steps:

- *Step 1*. We build a BSP of the scene. This subdivision already provides a cell-and-portal graph but it is too complex to be directly used in portal based applications. We call *bsp-cells* the leaves (convex polyhedra) of the BSP tree, and *bsp-portals* the transparent facets of each bsp-cell. We say that two bsp-cells are *connected in the BSP* if they share a common adjacent bsp-portal.

- *Step 2*. We build *valid separators* out of the set of bsp-portals we can find in the BSP subdivision. Valid separators are large portals made of smaller bsp-portals (see below).

- *Step 3*. The cells are built by looking for connected sets of bsp-cells: two bsp-cells are connected if they are connected in the BSP and the bsp-portal between them is not part of a valid

separator. See figure 2 for a simple example. After this step, we obtain a CPG with only valid separators.

- *Step 4*. We post-process cells to satisfy the constraint of the maximum rendering cost. If the polygons of the scene do not generate good portals, this step allows to add new separators that do not have to be aligned with the geometry.

The result is a cell-and-portal graph whose cells have a rendering cost under the upper bound defined by the user. Finally we apply the *simplification algorithm* on the CPG. It uses a metrics on portals in order to keep only the most relevant ones. The simplification algorithm can be applied to any CPG and can therefore be used as a post processing tool with any other cell-and-portal decomposition algorithm.

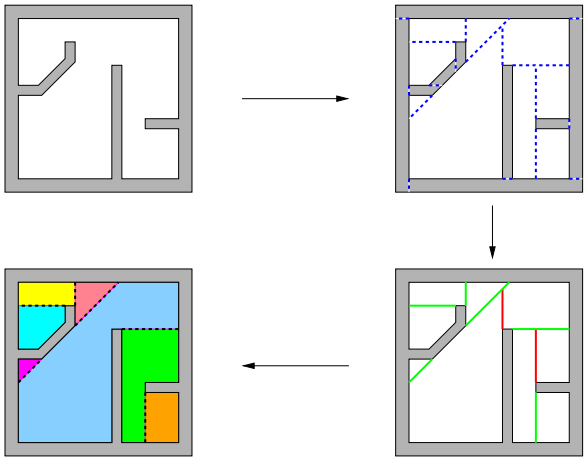

Figure 2: *Following the arrows.* **1**: input scene. **2**: BSP tree. **3**: valid (green) and non valid (red) separators. **4**: the resulting cell-and-portal graph at the end of the grouping algorithm. The simplification algorithm will merge adjacent cells, according to some user-defined constraints.

We want to find a set of portals that divide the scene into cells respecting user-given bounds: a lower and upper bound on the rendering cost of each cell. There are many ways to define a *rendering cost* for one cell. This include, for example, the number of polygons in the cell and their area, weighted by the cost of the hardware shader used to render each polygon. The cost of a hardware shader must take into account whether it uses multi-pass rendering, complex vertex-programs and fragment-programs (on modern graphic hardware) [11], etc... We call these bounds the *constraint* on the cells.

Our method outputs a **simplified** cell-and-portal graph, whose cells satisfy this user-given constraint. Figure 3 shows a typical cell created with our method.

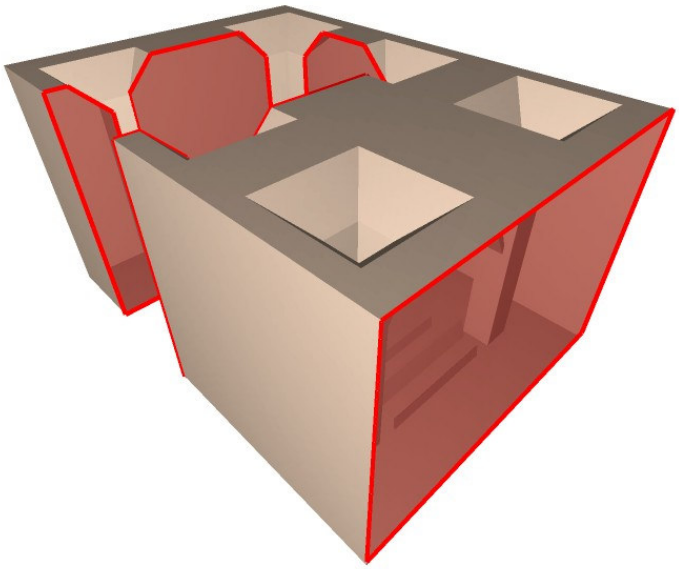

Figure 3: A cell created with our method. Portals are outlined in red.

2.1 The grouping algorithm

Step 1: initial BSP

We compute an initial BSP of the scene. It consists of a set of bsp-cells connected by bsp-portals. See section 3.0.2 for details. This initial subdivision of the scene *is* a cell-and-portal graph.

Let $\mathcal{C}_{init}$ (resp. $\mathcal{P}_{init}$) be the set of bsp-cells (resp. bsp-portals), created with the initial scene subdivision.

Let $\mathcal{G}_{init}$ be the graph $(\mathcal{C}_{init}, \mathcal{E}_{init})$ where $\mathcal{E}_{init} = \{(x, y) \in \mathcal{C}_{init}^2 : \exists p \in \mathcal{P}_{init}, p$ is a bsp-portal between bsp-cells x and $y\}$

$\mathcal{G}_{init}$ is the graph of bsp-cells connected by bsp-portals. It is the result of the BSP-tree computation and the input of the grouping algorithm. Note that there is a one-to-one correspondence between elements in $\mathcal{P}_{init}$ and elements in $\mathcal{E}_{init}$.

Step 2: building valid separators

We call polygons in the initial scene *solid* polygons. A desired property for a separator is that all its edges lie entirely on solid polygons of the scene. This is why we call them separators – as they are all supported by solid polygons, they hermetically close some pathway in the scene.

To build the set of all separators, we first group the bsp-portals according to their supporting plane.

Then within each plane, a connected set of bsp-portals defines a separator. Each separator now has to be checked for validity, which corresponds to the desired property stated above, see figure 4 (left and middle).

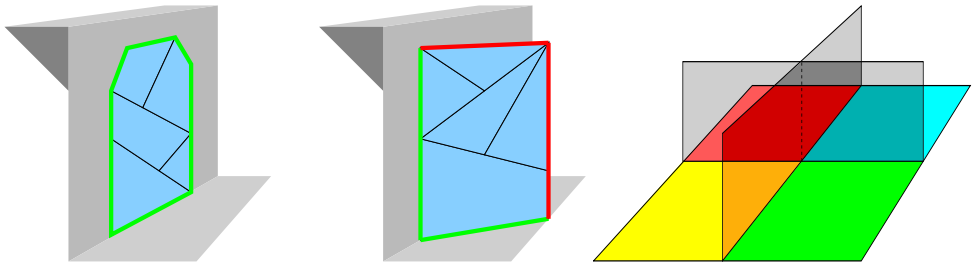

Figure 4: **Left:** 5 bsp-portals(blue) form a valid separator. **Middle:** the separator is not valid because of 3 non valid edges. **Right:** two crossing separators generating four cells.

More formally, a separator is *valid* if all its bsp-portals are valid. A bsp-portal of a separator is valid if all its edges lie on another bsp-portal of the same separator or lie on solid polygons of the scene. We call such edges *valid* edges: an edge is valid if all its points are adjacent to some solid polygon. Therefore, if one edge of a bsp-portal is not valid, the bsp-portal is not part of a valid separator: we remove this bsp-portal from the separator it belongs to, and check again for the validity of the separator.

Checking the validity of a separator with p bsp-portals has an $O(p^2)$ complexity, where p is generally small. After each removal of a bsp-portal, we also need to check if the bsp-portals of the separator are still connected. If not, we simply create a new separator for each connected component. The complete algorithm for building and validating planar separators is given in Figure 5.

After creating all separators we have to check if some of them cross each other. Actually if two valid separators cross each other this results in four non valid separators in the final cell-and-portal graph (see figure 4, right). If desired these crossing separators can be easily detected and deleted after cell creation (see step 3) as they connect more than two cells on one of their side. In this case, we need to choose a separator to be removed and repeat the process until no two separators cross each other. A simple way is to select the separator involved in the least number of cells. We can also use the metrics presented in section 3 to select the worst portal and remove it.

Step 3: creating the output cells

Now that we have valid separators placed in the scene, we need to build the corresponding cells. We process as follows: A bsp-cell is chosen and expanded by merging it with its adjacent bsp-cells; the cell stops growing when it encounters valid separators or solid polygons. We repeat this step while there exist some bsp-cells that have not been expanded into a (final) cell yet.

More formally, the important point is that a valid separator corresponds to a subset of $\mathcal{E}_{init}$ in the graph $\mathcal{G}_{init}$. Let $\mathcal{E}_{valid}$ be the union of such subsets: $\mathcal{E}_{valid}$ gathers all the bsp-portals that are part of a separator. We build the cells by looking for connected components in $\mathcal{G}_{init}$. Since the valid separators are new frontiers for the cells. We simply subtract $\mathcal{E}_{valid}$ from $\mathcal{E}_{init}$ before

```
group bsp-portals by planes;
FOR each plane,
  FOR each connected set of bsp-portals,
    build a separator;
    flag separator as not valid;
(*) WHILE separator not empty and not valid,
      FOR each bsp-portal P of separator,
        IF P is not valid,
           remove it from separator;
           create new separators if needed;
           goto step (*);
      IF all bsp-portals were processed,
        flag separator as valid;
```

Figure 5: Building and validating planar separators.

computing the connected components: each cell we construct is a connected component of the graph $(\mathcal{C}_{init}, \mathcal{E}_{init} \backslash \mathcal{E}_{valid})$.

Step 4: satisfying the upper-bound of the constraint

At this point we have a CPG with only valid separators. Before using our simplification algorithm on it, we have to make sure that all the cells satisfy the upper-bound of the constraint. Indeed the simplification algorithm tries to maximize the size of the cells by removing some portals. If the cells of the input CPG do not respect the upper bound constraint, then the cells of the output CPG will not respect it either.

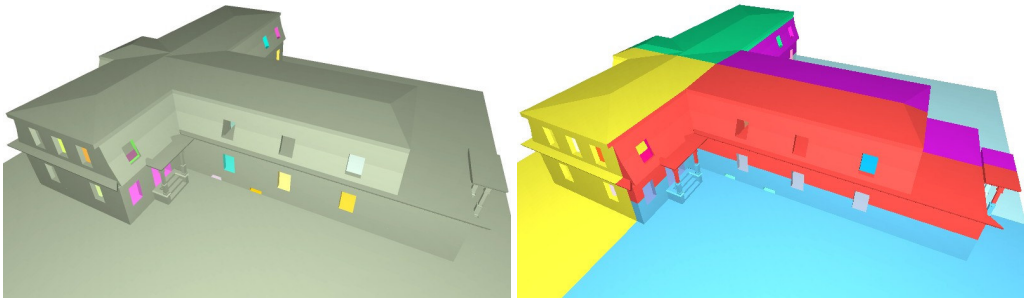

Figure 6: **Left:** In this big house, The exterior (gray) cell is too big. **Right:** It is split to respect the constraint.

Generally, a cell is too large when no cutting plane in the BSP generates good separators in it. However, the cell is a closed polyhedra whose facets are solid polygons or portals (the separators). Therefore we can choose some cutting plane and cut the cell by this plane in order to create a new separator. This new separator need not be valid (as defined above) but we will keep it because we

need the cell to be split into smaller ones. We do not consider this new plane as a cutting plane for the other cells. It is created using simple heuristics, detailed below. Figure 6 (right) shows how the gray cell (left) has been split. In this example, splitting the exterior cell is useful to avoid drawing the whole exterior when the camera is inside and look outside, through a window.

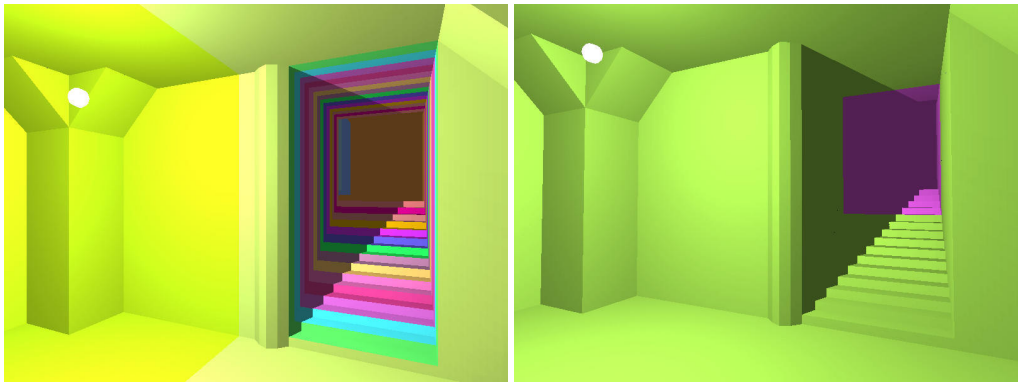

Figure 7: **Left:** Before merging cells, there are many cells in the stairs of the Clinic scene. **Right:** After merging, only two cells remain and are visible.

Split heuristics. When we need to split a too large cell, a set of planes is created using an heuristic. The plane that creates the best balanced cells is chosen as the new cutting plane: the idea is to cut a big cell in two equally-sized smaller cells. Figure 10 shows the results with different heuristics for the creation of the cutting planes:

- Choose a random point inside the cell and a random normal for the plane: this gives good results in terms of cell size, but the shape of cells becomes more complex. Figure 10 (left).
- Same as above, but take an axis-aligned normal: it gives better cell shapes, but is limited to axis aligned splits. Figure 10 (middle).
- Randomly choose a bsp-portal included in the cell that is not part of a separator: it gives the best results as bsp-portals corresponds to cutting planes in the original BSP tree. Figure 10 (right).

3 The simplification algorithm

The purpose of this algorithm is to simplify a CPG by removing portals. The algorithm stops when the rendering cost of all cells is below the maximum rendering cost constraint and preferably higher than the minimum rendering cost. As explained before this requires all cells of the input CPG to be under the upper bound of the constraint.

We first sort cells according to their rendering cost. Then we examine the separators of the smallest cell, excluding separators that would yield, if deleted, a cell that exceeds the rendering cost upper bound. Remaining separators are compared using the metrics described in section 3 and the one with the higher score is removed, while its two adjacent cells are merged, and the sorting of the cells updated. We iterate that process until the constraint is respected by all cells. Figure 7 (right) shows how the stairs' steps have been merged into a larger cell. Notice that in some rare cases we may not be able to merge a small cell to one neighboring cell. So the lower bound on the rendering cost for each cell may not always be satisfied by all cells, whereas the upper bound is guaranteed. This happens if all neighboring cells' rendering cost are near the upper bound.

We have now obtained the final cell-and-portal graph. The CPG respects the constraint while having good geometrical properties (e.g. simple and large separators). Sequence 5 of the accompanying video shows the simplification algorithm in action.

In order to choose the portals when simplifying a cell and portal decomposition, we need a metrics to measure the efficiency of a portal. We propose such a metrics, which is computed using graphic hardware.

3.0.1 What is a good portal ?

Defining what is a good portal is not an easy task. We suggest that a good portal should optimize two main properties:

1. it must be as hidden as possible: we want to minimize the volume, inside the scene, from which the portal is visible (the *visible volume* of the portal).
2. it must be good at separating *the rendering cost*: we want the portal to reject a lot of geometry when it is not visible, on both sides. Portals inducing neighboring cells with nearly the same rendering cost are a better choice than portals with a big cell on one side and a small cell on the other.

Note that even if a good portal's placement often corresponds to the intuitive portal placements (doors, windows, ...) this is not always the case. The figure 8 shows a cell at different steps of the simplification algorithm.

Figure 8: A local view of a CPG during the simplifi cation algorithm. Portals are in white with a blue outline. Our metrics is used to determine which portal should be removed at each iteration. Notice that the last portal corresponds to what we defi ne as a good portal: its neighbouring cell costs are well balanced and it is not in an area easily visible from the scene (it is less visible from the lower floor than the portal at the top of the stairs in picture 3).

3.0.2 Computing the metrics using graphic hardware

Visible volume. In order to have a fast evaluation of the visibility volume we rely on the graphics hardware. The low precision is not an issue since we only want to distinguish a good portal from a bad one. The precision should only be sufficient to sort the portals. We evaluate the metrics at one point. To compute a portal's efficiency, we average the values evaluated at sample points on it. As we need to have information from all directions around a sample point, we rely on the rendering of cube-maps, on low resolution viewports (typically 32x32 pixels). Each pixel's depth value in the cube-map gives us an estimation of the visible volume seen through that pixel (with the appropriate view angle correction). We sum the contribution from all pixels in the cube-map to estimate the visible volume at one point.

Rendering cost balance. As we want to know if the rendering cost is well balanced on both sides of a portal p, we compute

$$balance(p) = |cost(cell_A(p)) - cost(cell_B(p))|$$

where $cell_A(p)$ and $cell_B(p)$ are the cells sperated by the portal p. The rendering cost of a cell is the one defined in 2.

Comparing portals. We are looking for portals minimizing both the $balance()$ and $visibleVolume()$ functions. Minimizing their sum is not a good idea, since it would need additional work to weight their respective contribution to the sum. For a portal p, we define the metrics

$$score(p) = visibleVolume(p) \cdot balance(p)$$

As stated earlier the choice of a BSP-tree as the initial subdivision seems mandatory. Imagine for example that we use a Delaunay tetrahedralization to compute an initial subdivision. The good thing is that all (parts of) the input polygons correspond to a triangle of two adjacent tetrahedra of the subdivision. The bad thing is that finding two coplanar Delaunay-portals almost never happens. Therefore we will not be able to build good separators.

Using a BSP tree makes sure that many bsp-portals are coplanar and can be grouped into valid separators.

A better subdivision for our method would be the arrangement of the supporting planes of the scene polygons. Indeed all separators aligned with a polygon would be found. Having more separators reduces the size of the biggest cells and tends to limit the number of non valid separators introduced by the splitting step. In practice however, the size of this arrangement, $O(n^3)$, is too high to be used with large scenes.

To adapt to different types of scenes, we can guide the beginning of the BSP tree construction as follows: instead of using a heuristic at the *beginning* of the BSP tree construction, we can explicitly specify the cutting planes, as done when a scene is cut with a regular 3D grid. In each such "regular" bsp-cell (e.g. a 3D grid cell), the rest of the BSP tree is computed as usual (with a heuristic to find the next cutting plane). A 3D grid gives bsp-portals and separators aligned with the guiding grid's planes, while simple BSP trees produce separators supported by polygon planes. A regular tetrahedral grid gives separators with four possible orientations. The user may also want to give pre-defined planes or boxes whose facets the BSP construction should consider as the first cutting

planes. We believe the user should choose a space subdivision depending on the type of scene they are designing.

Bsp-portals are explicitly created during the BSP computation by clipping the current cutting plane on parents cutting planes and recursively splitting these bsp-portals by children cutting planes [4].

Heuristics for BSP construction

The choice of the cutting plane during the BSP construction is based on a heuristic. For a general scene with n input polygons, the BSP tree construction complexity is $O(n^3)$ in time and $O(n^2)$ in space, in the worst case. Some heuristics ensure an $O(n)$ expected space. Others accelerate the processing time to construct the BSP.

We refer to the papers by Airey and Teller[1, 14] for discussion on various heuristics and their efficient combination.

The problem of finding an optimal subdivision for our grouping algorithm is ill conditioned. We do not know how to define an optimal solution, since we don't know how to compare two acceptable solutions. However the more portals the subdivision creates, the more choice the grouping algorithm has. Moreover we do not need to have a well balanced tree since this tree is not used to perform fast searches. We experimented with the following heuristics:

- **min–cut** minimizes the number of cut polygons. With k polygons in the subspace under consideration, it takes $O(k^2)$ times. The BSP in constructed in $O(n^3)$ time.
- **max–area** chooses the planes that has the greatest area covered by polygons. If the supporting planes of input polygons are initially sorted by area (which takes $O(n \log n)$), with k polygons in the subspace under consideration, **max–area** takes $O(k \log k)$ time.
- **max–ortho** chooses the planes that maximize the orthogonality between the plane and the cut polygons.

Whatever heuristic is used, our method performs well. That is, it always outputs a usable cell-and-portal graph. Nevertheless the min–cut heuristic gives the most satisfying results, because it tends to place details (tables, chairs, fire-places, computers, showers, wash-hand basins) entirely into one cell, which is visually pleasing.

Some specific scenes may require special guidance at the beginning of the BSP tree computation: for example, for cave-like scenes (blood vessels, airway, ...) where no polygon is oriented perpendicularly to the local cave's orientation, simple BSP are really not suited and one should rather try a guided BSP subdivision, as described earlier. The other option would be to rely on the automatic splitting of too large cells to handle such cases, provided that a convenient split heuristic is used.

In our implementation, we use a brush[5]-based BSP tree to create the initial cell-and-portal graph when the scene is modeled as a set of brushes. Brush-based BSP trees are widely used in the game developer community to compute PVS in game levels. This allows us to use a standard game editor output as an input for our algorithm. The main advantage of brush-based BSP is that it allows a robust interior-exterior test during the BSP tree construction. Therefore interior bsp-portals and

[4] Using an algorithm similar to the one used by `qbsp` from ID software

[5] A *brush* is a convex polyhedron.

interior bsp-cells can be detected and deleted. Nevertheless, our implementation easily handles polygon soups (sets of independent polygons with no connectivity information) by using a different algorithm for the creation of the portals. However if the input is not watertight it is not possible to distinguish the interior from the exterior and therefore cells are created "inside" the model. This results in a longer computation time when dealing with non watertight models.

The BSP construction is processed completely for two reasons: suppose we stop the BSP construction when each leaf contains few polygons (e.g. 10, to speed up the processing of a scene). *First*, we lose many bsp-portals that could be crucial to obtain valid separators. *Second*, we cannot perform an interior-exterior test anymore, which forces to keep all undesirable bsp-cells created "inside the walls".

Since the algorithm requires many geometrical tests, precision issues can occur. To reduce precision problems when grouping bsp-portals by plane, each cutting plane is assigned an ID when computing the BSP tree. When cutting a sub-space all polygons and portals belonging to the cutting plane are given the same ID. Thus we can group bsp-portals according to their ID. Note that the same plane can be a cutting plane in two different sub-spaces. To handle that case we have no other choice than using a numerical comparison of the planes to find if the cutting plane already has a coplanar plane. In this case we assign them the same ID. Robustness issues are handled with some small ϵ value. The same ϵ precision is used to check if a bsp-portal edge lies on a polygon of the scene during separator validation.

In fact it should be possible to build a connectivity structure on the portals and polygons during the BSP computation in order to avoid most precision problems. This was not fully implemented because the ϵ-precision we use works well in practice. However, the connectivity structure should speed up the validation of separators in a significant way, as we could check in constant time whether an edge is valid or not.

Figure 9 (left) shows the cells created with our method for a simple scene.

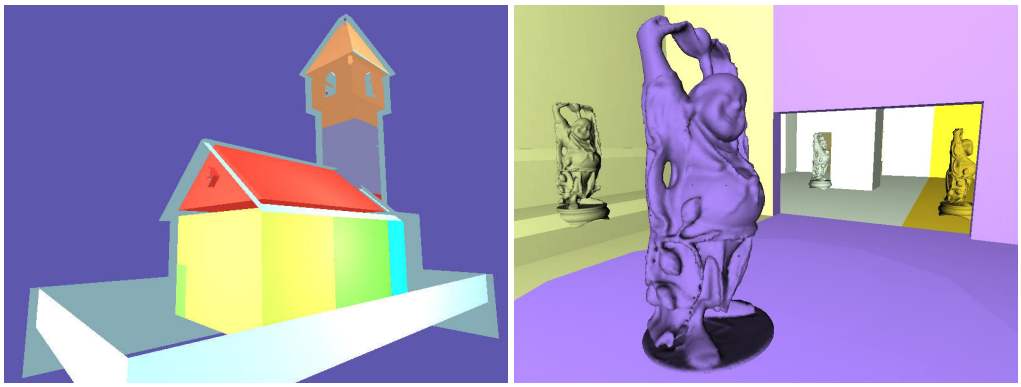

Figure 9: **Left:** A church is subdivided into cells. Each color represent one cell (for "clarity", back-faces are rendered). **Right:** We placed one buddha in each cell in order to build a scene with 820,000 polygons, and about 16,000 polygons per cell.

Table 1 presents timings for the BSP tree computation, using the **min-cut** heuristic. The number of bsp-cells is roughly the same as $\diamond out$. Note the very high number of portals created, making it useless for portal rendering. The accompanying video shows (in sequence 1) a captured walk-through in the clinic model, highlighting the complexity of the bsp-cells.

The timings were done on an Athlon@1466 Mhz, with 1Go of RAM.

Scene name	**# in**	**BSP**	**# out**	**$\diamond$ out**
Church	1056	3 s.	3523	2460
Underground	2105	3 s.	4972	3248
Sanatorium	4916	8 s.	10501	8314
Clinic	11597	135 s.	27241	16816
Blockwar	21369	123 s.	47337	31222

Table 1: Statistics for the BSP construction. Column **# in** gives the number of input polygons; **BSP** gives the time needed to compute the BSP tree; **# out** shows the number of polygons in the BSP tree, and $\diamond$ **out** shows the number of portals in the BSP tree.

Table 2 shows statistics on the grouping algorithm, performed on the same machine. Notice the drastic reduction in the number of cells after the grouping algorithm (**# cells**, left), and the effect of the lower bound of the constraint (**# cells**, right). The video shows (in sequence 2) the same walk-through after applying our method to the model, clearly demonstrating the usefulness of the new decomposition.

Scene name	validation	# cells	# portals
Church	2 s.	68/15	18
Underground	5 s.	97/20	13
Sanatorium	19 s.	186/18	33
Clinic	66 s.	630/141	103
Blockwar	176 s.	869/177	94

Table 2: Statistics for the grouping algorithm, with a merging threshold of 50 polygons and no splitting threshold. Column **validation** shows the time needed to compute the valid separators and to construct the final cell-and-portal graph (although the time for constructing this graph is negligible). **# cells** shows the number of cells after the grouping algorithm (left) and the number of cells after merging tiny cells (right); **# portals** shows the number of final portals (valid separators).

To demonstrate the usability of our decomposition method, we present the rendering speed of walk-throughs in some of our models (see table 3). The tests were made with a GeForce4Ti graphic board, on the same computer as above. The rendering is done using a *per pixel* Phong shading: the rendering time of such a hardware shader is about twice higher than classical texture mapping with OpenGL standard lighting.

The column **BSP** simply illustrates how BSP are really not suited to portal rendering. Too much time is spent checking whether portals are visible, because there are too many small and mutually

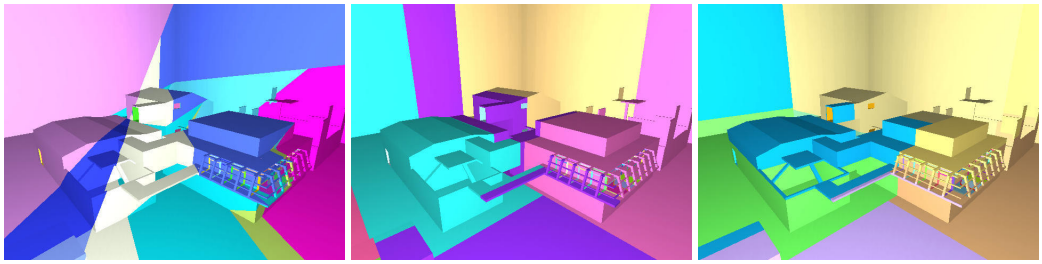

Figure 10: Different criteria for cell split. From left to right, random choice, random choice aligned on axis and random choice on unused portals

visible portals. Moreover, each cell contains too few polygons, so that the graphic pipeline is stopped way too many times.

Column **our graph** demonstrates the usability of the cell-and-portal graph computed with our method for portal rendering. We have much less portals to handle, and each cell is large enough. This allows to send big chunks of geometry to the graphic board while not hashing the graphic pipeline too much.

The last line of table 3 shows statistics for a large scene. For that purpose, we placed a buddha model (15536 polygons. figure 3, right) in each cell of the Clinic model, obtaining a scene with 820,000 polygons and nearly equal-sized cells (about 16,000 polygons each). As usual, furniture (the buddha model) is inserted into the cells after the computation of the cell-and-portal graph, as in [14]. The results demonstrate the interest of having larger cells for *on-line* visibility determination; each cell can contain detailed objects that are not drawn if the cell is not visible.

Scene name	BSP	all	our graph
Clinic	7-20-39	20-30-33	75-103-149
Blockwar	3.7-6.3-6.5	18-22-23	87-126-171
Clinic+buddhas	–	– -1.2-1.3	5-18-34

Table 3: Rendering speed in frames per second (minimum, average, and maximum along a walk in the scene). In column **BSP**, we use portal rendering on the initial BSP subdivision. In column **all**, we simply send all scene polygons to the graphic hardware. In column **our graph**, we use portal rendering with the decomposition computed with our method. The clinic model with additional buddhas contains approx. 820,000 polygons.

4 Conclusion, future work

We have described a general tool that *automatically* computes a usable cell-and-portal graph for hardware accelerated interactive rendering, from a polygonal scene.

The method is flexible in many ways. Its input is a BSP of the scene, that can be of various kind to accommodate the style of the scene: Standard BSP or brush-based BSP, possibly guided using 3d-grids or other guiding schemes. However, the output may be unsatisfying for some reasons: A portal placed where the artist did not want one, portals missing in other places due to the initial subdivision or too large cells. A user interface should be implemented allowing the user to:

- Choose the method to create the initial subdivision: guided or not, guided with boxes, grid,...
- Delete a portal. To delete a user-selected portal, one could simply merge its adjacent cells C_1 and C_2, and delete the other portals binding C_1 and C_2.
- Add a portal. With a user-defined polygonal portal, one could re-cut the BSP-leaves where needed and rebuild the neighbors cells by looking at the new connected component in the graph.

One would like to use a faster decomposition method. We believe that the metrics introduced in this paper to evaluate the efficiency of a portal could be used earlier in the method; for instance, to help the construction of the BSP tree. Our simplification algorithm can also be used on any CPG, allowing to adapt it to modern hardware constraints (i.e. optimizing the use of bandwith). Nevertheless, the notion of good CPG shoud be explored deeper. Finally, performance comparison between "portal rendering" on models processed with our method, and other occlusion culling method should be done.

References

[1] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations.* PhD thesis, Dept. of CS, U. of North Carolina, July 1990.

[2] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):41–50, March 1990.

[3] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In ACM, editor, *Proc. Symposium on Interactive 3D Graphics (SI3D'97)*, pages 83–90, 1997.

[4] F. Durand, G. Drettakis, and C. Puech. Fast and accurate hierarchical radiosity using global visibility. *ACM Transactions on Graphics*, April 1999.

[5] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structure. *Proceedings of SIGGRAPH'80*, 14(3):124–133, 1980.

[6] A. James and A. M. Day. The hidden face determination tree. *Computers and Graphics*, 23(3):377–387, july 1999.

[7] C. B. Jones. A new approach to the 'hidden line' problem. *Computer Journal*, 14(3):232–237, Aug. 1971.

[8] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In A. Press, editor, *Proc. ACM Symposium on Interactive 3D Graphics, special issue of Computer Graphics*, pages 105–106, Monterey, CA, April 1995.

[9] D. Meneveaux, E. Maisel, and K. Bouatouch. A new partitioning method for architectural environments. Technical Report 1096, IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France, April 1997.

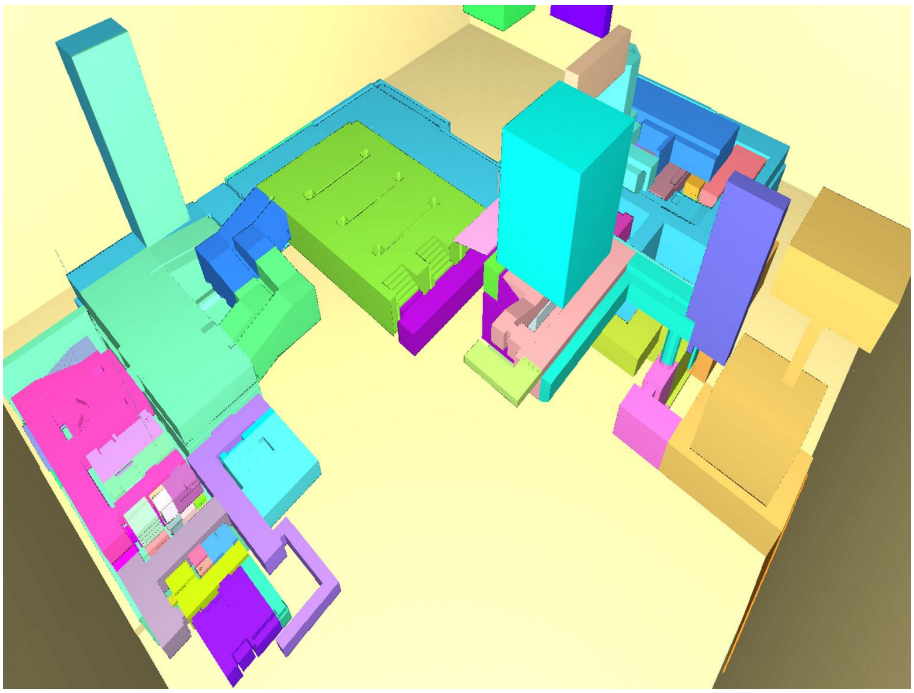

Figure 11: One view of some game's scene. Each cell has a different color. Here, the constraint demanded a high number of polygons per cell. Note the way stair's steps are captured in a single cell.

[10] D. Meneveaux, E. Maisel, C. F., and K. Bouatouch. Partitioning complex architectural environments for lighting simulation. Technical Report 2981, INRIA/IRISA, 1996.

[11] nVIDIA. http://developer.nvidia.com/. web pages.

[12] nVIDIA. http://www.nvidia.com/dev_content/nvopenglspecs/ GL_NV_occlusion_query.txt. web page.

[13] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete & Computational Geometry*, 1989.

[14] S. Teller. *Visibility computation in densely occluded polyhedral environments*. PhD thesis, UC Berkeley, CS department, 1992.

[15] S. Teller and P. Hanrahan. Visibility computations for global illumination algorithms. In *Proc. Computer Graphics (SIGGRAPH'93)*, volume 27, pages 239–246. ACM Press, July 1993.

[16] S. J. Teller and C. Séquin. Visibility preprocessing for interactive walkthroughs. In *Proc. Computer Graphics (SIGGRAPH'91)*, volume 25, pages 61–69. ACM Press, 1991.

[17] M. van de Panne and J. Stewart. Efficient compression techniques for precomputed visibility. In *Proc. Eurographics Rendering Workshop*, pages 305–316, June 1999.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399