Estruturas Espaciais

Introdução

- Estruturas espaciais são estruturas criadas para organizar dados em regiões semânticas através de planos de corte

- Dados espaciais ocupam regiões do espaço (sua extensão), definidas por sua posição e limites

Introdução

- A diferenciação de estruturas espaciais se dá principalmente pela forma como os índices de classificação são formulados

- Exemplos de estruturas espaciais incluem as **kd-trees** e as **BSP-trees**

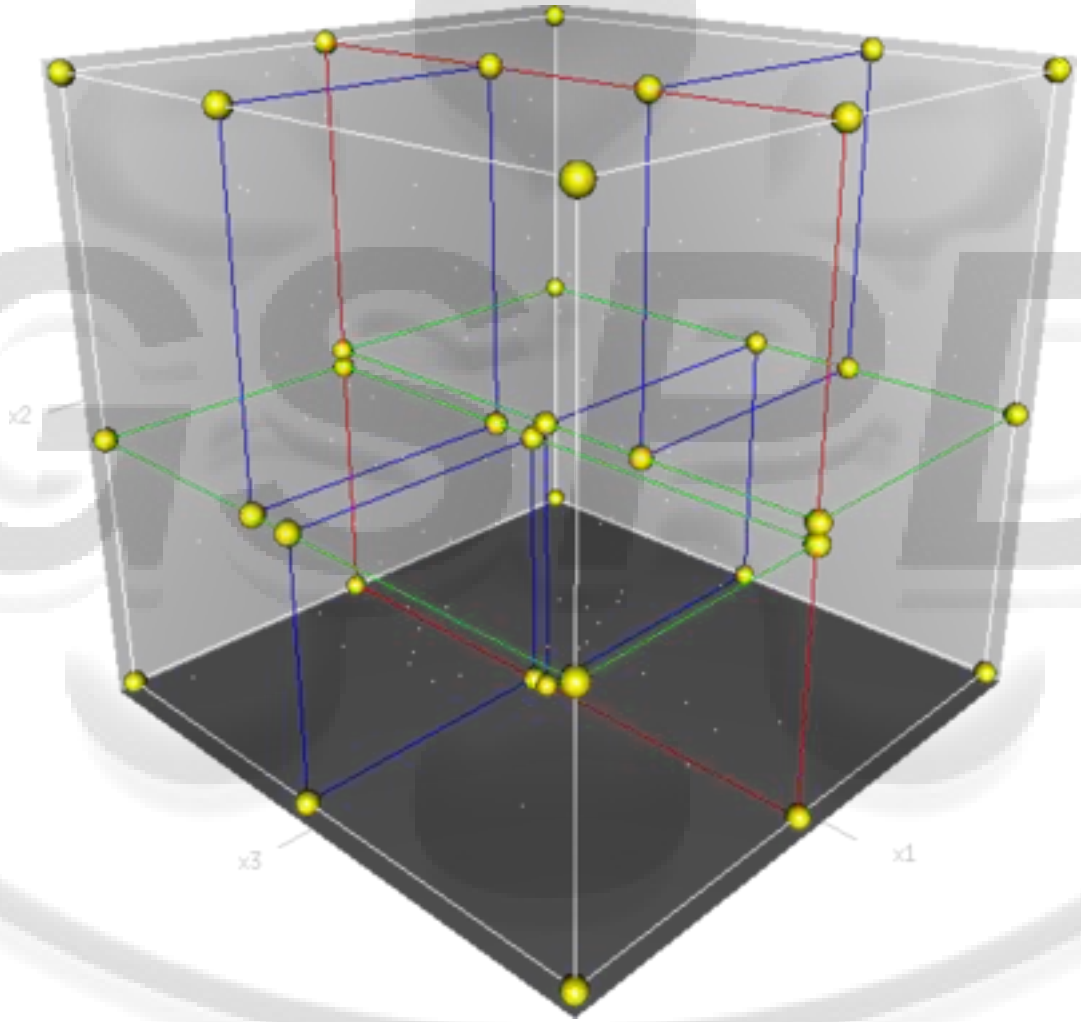
Árvores kd (ou kd-trees)

- São árvores em que cada nó representa uma região retilínea
- Nelas cada nó está associado a um plano, que corta um dos eixos de forma longitudinal
- Em geral os cortes alternam os eixos a serem cortados de acordo com a profundidade na árvore

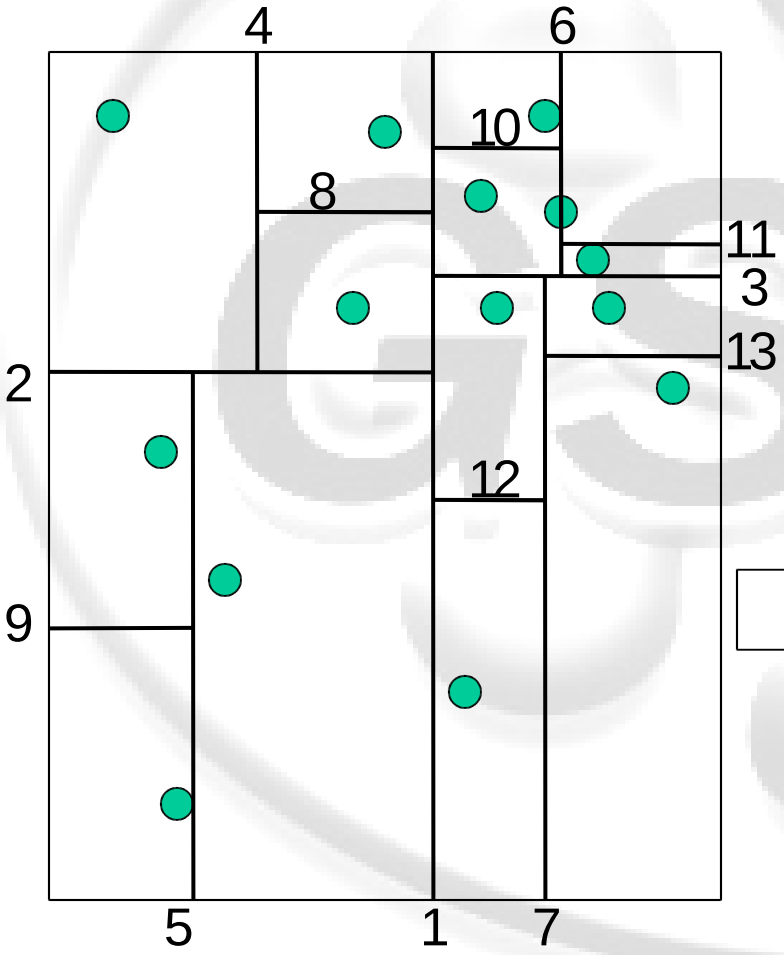
Árvores kd

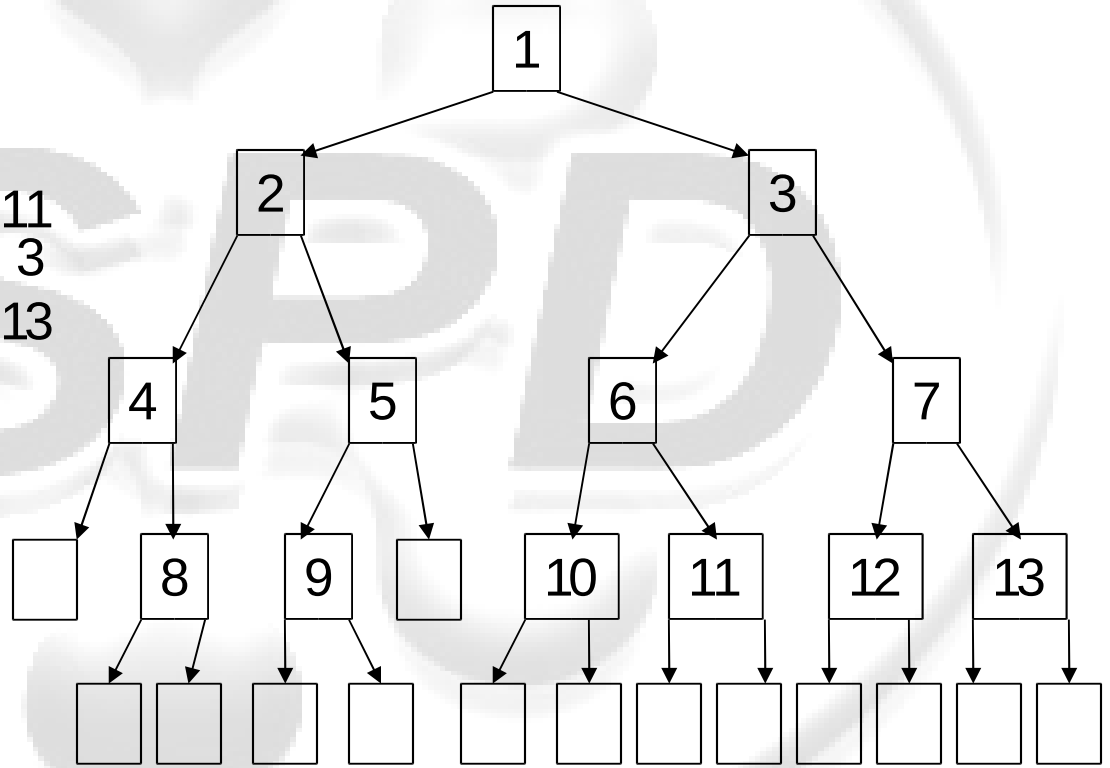
- Embora kd implique em k dimensões, boa parte das aplicações acaba restrita a três dimensões
- Árvores nessa situação são também conhecidas como árvores 3-d
- As figuras a seguir mostram exemplos de árvores 3-d e 2-d

Árvores kd



Árvores kd





Construíndo árvores kd

- A estruturação de uma árvore kd está baseada, fundamentalmente, na separação dos elementos da árvore através de planos de corte

- Assim, temos que definir como esses planos de corte são efetivamente criados

Construíndo árvores kd

- A abordagem canônica para a construção de uma árvore kd envolve:

1) Ordenar os dados
2) Escolher o plano de corte sobre o elemento mediano
3) Repetir os cortes sobre os pontos medianos das regiões resultantes

Conteúdo de um nó

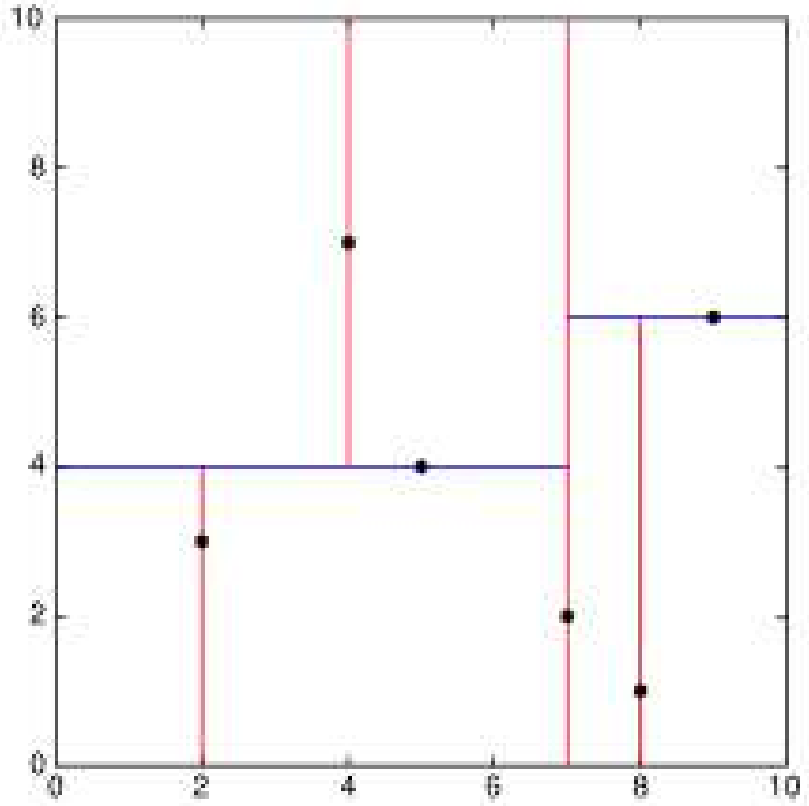
- Ponteiros para nós filhos (sempre dois)
- Extensão da célula (coordenadas máximas e mínimas em cada eixo)
- Os dados (pontos) contidos na célula
- Possivelmente incluir
  - Apontador para nó pai
  - Apontadores para vizinhos

Exemplo

- Construir a árvore 2d para os seguintes pontos:

```
[(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)]
```

Exemplo



Aplicações de árvores kd

- Envolvem situações em que a separação dos dados em regiões é importante, como:
  - Jogos em 3D
  - View frustrum culling
  - Renderizações
  - Indexação espacial em BD

Árvores BSP (ou BSP-trees)

- Árvores BSP (Binary Space Partition trees) podem ser vistas como uma generalização de kd-trees

(ou estas como uma especialização de BSP-trees) ☺

- A diferença entre BSP e kd está na forma de inserção dos planos de corte

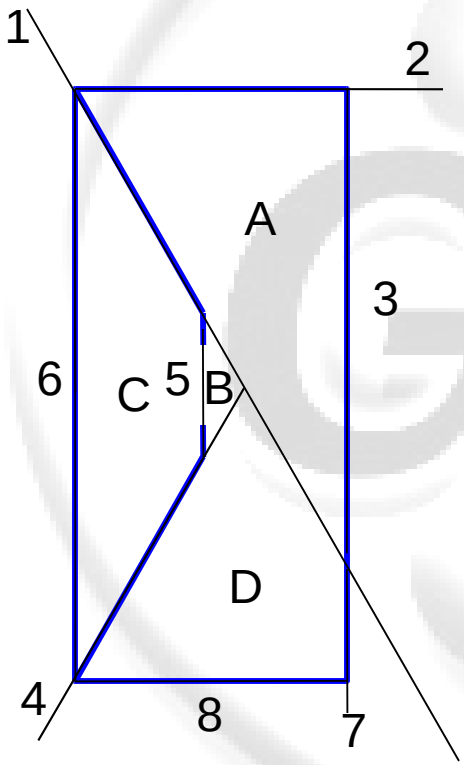
Árvores BSP

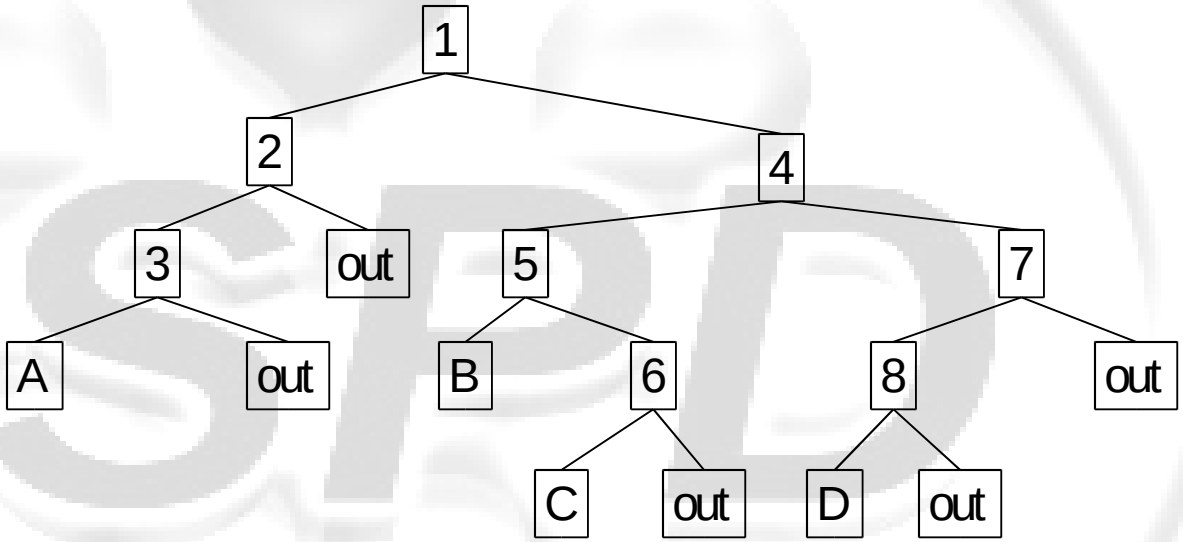
- Numa árvore BSP os planos podem interceptar os eixos em qualquer ângulo

- Cada plano termina ao encontrar o plano definido por seu pai

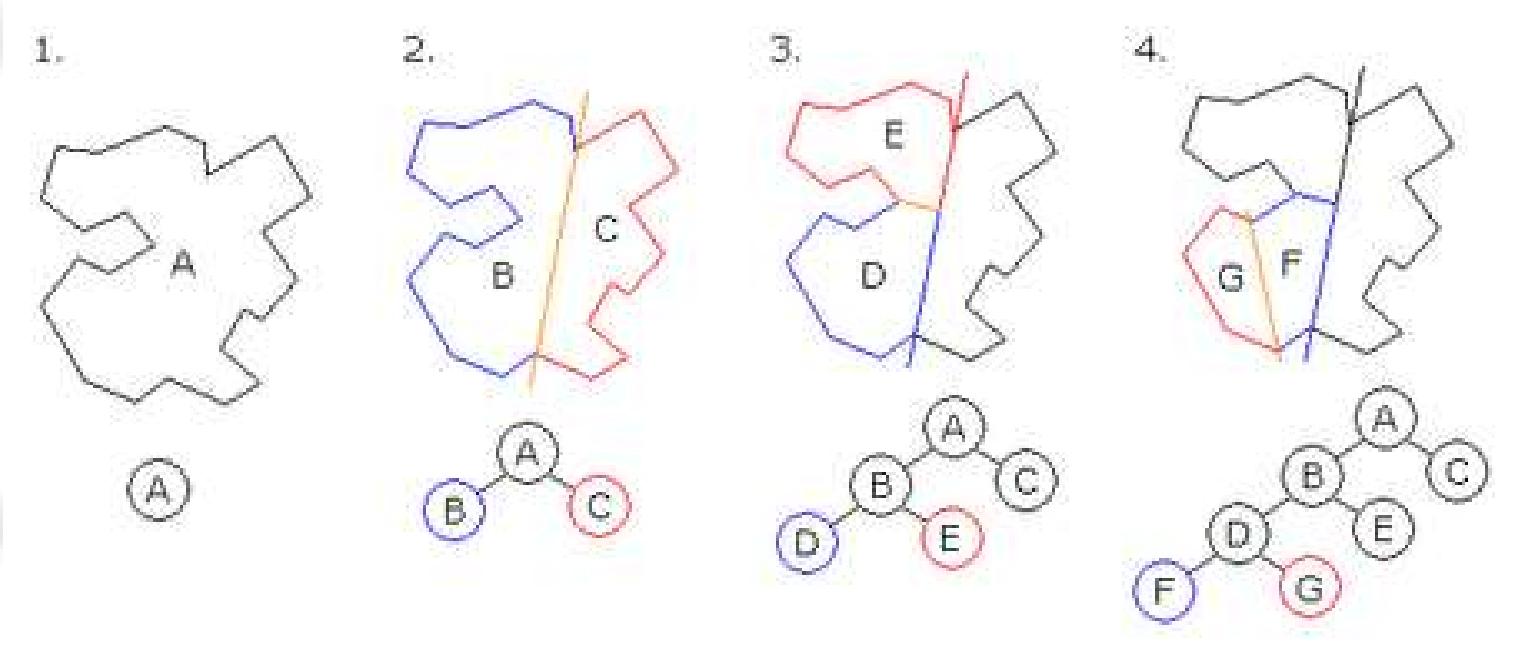
- Isso cria regiões convexas (polígonos), uteis em aplicações como jogos 3D

Árvores BSP





Árvores BSP



Construíndo árvores BSP

- Mais uma vez o problema está na escolha dos planos de corte

- As diretrizes básicas envolvem a minimização do número de polígonos gerados e a não divisão dos objetos entre polígonos vizinhos

Conteúdo de um nó

- Ponteiros para os nós filhos
- Sua extensão se for um nó folha
- O plano de divisão se for um nó interno
- Os dados contidos pela célula
- Possivelmente incluir
  - Ponteiros para o nó pai
  - Ponteiros para os nós vizinhos
  - Portais para visualizar dentro dos vizinhos

Aplicações de árvores BSP

- Árvores BSP são consideradas como a estrutura padrão geração de imagens 3D em jogos

- Um exemplo de aplicação é visto na geração de uma imagem do jogo “The wotch, my sister, and myself”

Árvores BSP

Step 1.

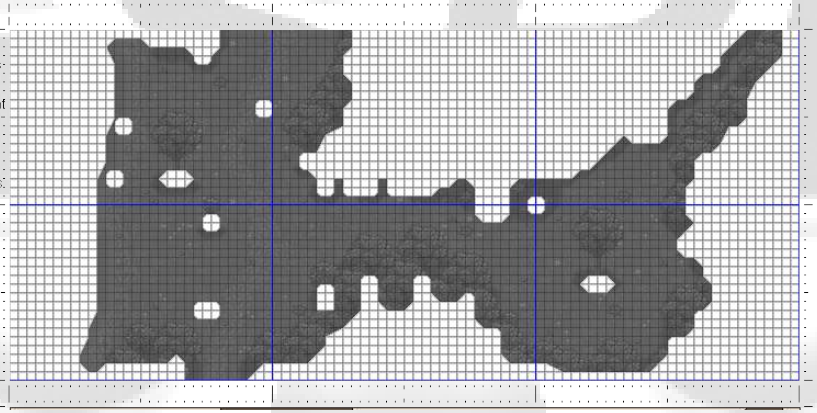
The level designer places tiles (little 8x8-pixel hand-drawn images) where she wants various things like grass and trees and rocks to appear in the room.

Our game allows multiple small tilesets to be loaded simultaneously; one tileset may contain a dozen tiles for trees, while another may contain tiles for grass, while still another may be an animated tileset for water.

Step 2.

The level designer switches from the tile layer to the "blocks" layer. The blocks layer is an invisible layer that the designer can draw on to indicate where the player is and is not allowed to walk.

Various block shapes are available, including squares and diagonals, but they're all just tiles just like in the main tile layer above.

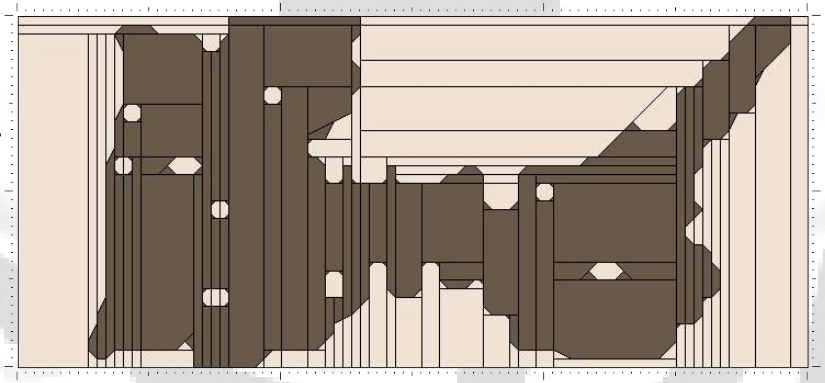


Árvores BSP

Step 3.

When the designer goes to save the game, the level-editing tool performs some magic behind the scenes. The block-tiles in the blocks layer are transformed into polygons, and merged with each other to create bigger polygons.

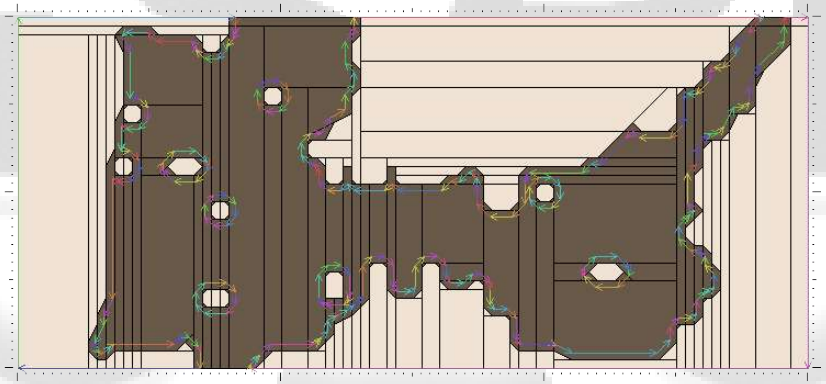
The merging process is performed by a BSP tree generator, and results in convex polygons (that's polygons that are only "round" --- there are no "dents" in them) for both "solid" areas, where the player cannot go, and "space" areas, where the player can go.



Step 4.

The level editor program continues its magic, and generates "expanded windings" for the polygons. The expanded windings are the same as the outer edges of the polygons, but "pushed" outward by exactly 6 units (pixels in the game world).

The resulting "windings" are merged together using a constructive-solid-geometry (CSG) union technique, resulting in large, nonoverlapping polygons that describe places where the player cannot walk (the full "expanded windings" are shown here composed of colored arrows.)

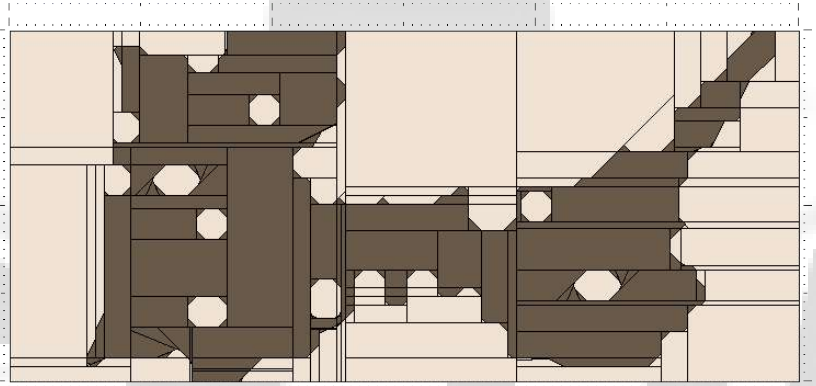


Árvores BSP

Step 5.

The level editor continues its magic by feeding the new windings to another BSP generator that then produces a set of "walkable" space polygons.

The reason for expanding the polygons is to avoid having to represent the player as a large circle in collision-detection. If, instead, the player is only a point thick, and the walls are "thicker," we get the same results, but with less computation. Comparing a point against a set of walls (a set of lines) is much easier and much faster than comparing a circle against the same walls.

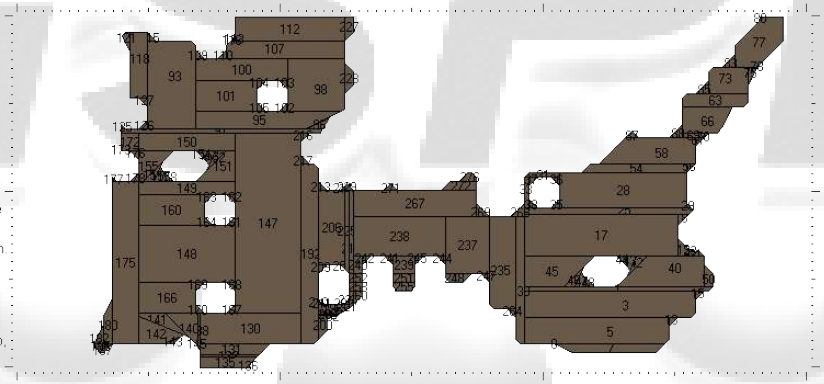


Step 6.

The "solid" polygons are discarded, and the "space" polygons are assigned unique numbers to identify them. The program computes a connectivity graph, and then computes all-points-shortest-paths on the graph.

The result of this is that the player can be easily checked to ensure that she is always standing inside one of the polygons, which provides collision-detection.

The connectivity graph can tell an enemy that if the player is standing in, say, space #73 here, and the enemy is in space #93, the enemy should next move to, say, space #95 if he wants to go toward the player.



This data is then compressed and saved into the output file for the game engine to use, along with the tiles, which are really nothing more than eye candy.

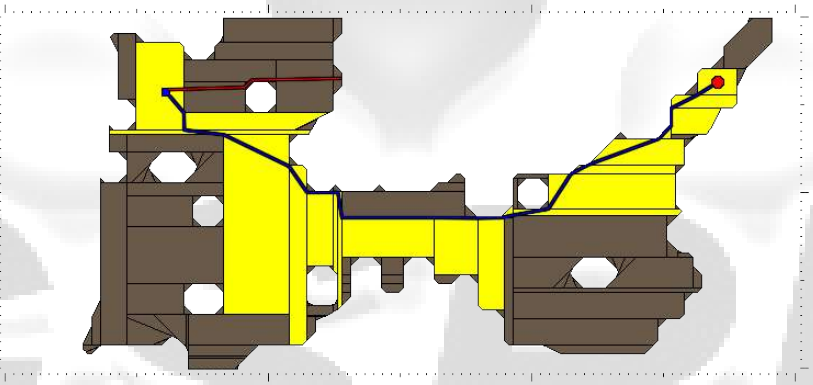
Árvores BSP

This data is then compressed and saved into the output file for the game engine to use, along with the tiles, which are really nothing more than eye candy.

Within the game engine, this shows how an enemy (the blue square) would try to seek the player (the red square) using a simple greedy pathfinding algorithm based on the connectivity graph. The yellow highlighted areas show the regions that the pathfinder would select for the enemy to pass through, and the blue line shows the actual path the enemy would follow.

While the blue path shown here is not perfect, it is very *close* to perfect, close enough that the player won't notice if the enemy occasionally turns strangely.

The red line shows the simple path an enemy would most likely take without the connectivity graph to help him.



Here, we can see the two possible paths superimposed over the tiles, and we can see how "intelligent" the enemy has become with the connectivity graph. No longer does he simply run toward the trees; but instead, he starts south!

Following his path, we can see how humans attribute his dodges to intelligence rather than to the greedy pathfinding; a human would assume that his first dodge east was simply to avoid the tree. He then skirts around the southern edge of the trees, zigzagging a little as he seems to "decide" on his path, and finally makes a bee-line for the player.