Occupancy-Regulated Extension
Using Chunks to Build Levels

Peter Mawhorter    Michael Mateas
Department of Computer Science
University of California Santa Cruz

August 25, 2010
Motivation

- Existing generators impose constraints in pursuit of a goal.
Motivation

- But human designers often create surprising levels.
The goal: create levels that can routinely surprise their creator.
Randomly placed components would be surprising, but not interesting.

Placing groups of components reduces entropy, and can exploit human authoring.

Occupancy can be used to constrain assembly of chunks.

Thus Occupancy-Regulated Extension.
The ORE Algorithm

1. Select a context.

2. Pick a chunk to insert:
   (i) Filter available chunks.
   (ii) Select among compatible chunks.

3. Integrate the selected chunk into the level.
Occupancy in ORE

- Occupancy is expressed as concrete anchor points.
- Each chunk defines its own anchor points.
- These anchors determine how pieces can fit together.
Context Selection

- Picks a random anchor at which to add a chunk.
- Keeps track of used and unused anchors.
- Handles edge cases: might reset the list of used anchors, or even improvise a new anchor.
Example

- The initial context:
Chunk Filtering

- Uses a notion of spatial compatibility to exclude things that don’t fit.

- Determines type compatibility for overlapping components.

- Filters out chunks that would extend outside of the bounding box of the level.

- Considers each chunk in the library at each of its anchors, so the algorithm isn’t directional.
Example

▶ An example library:
Example

- The matching anchors:
Example

▶ The first match:
The second match:
Example

- The third match:
Example

- One of the non-matches:
Example

- Another non-match:
Chunk Selection

- Considers only the first several (currently 17) filtered chunks.

- Computes chunk metrics:
  - $f$: Chunk default frequency, as defined in the library.
  - $b$: Chunk boredom value: number of times the chunk has been used so far.
  - $p$: Chunk precision bias: 0.2 if the chunk is labeled as “precise”; 1 otherwise.

- Calculates a weight for each chunk being considered:
  - $w = f \times 0.7^b \times p$
Chunk Selection

- Uses weighted random selection with the computed weights to choose a chunk to insert.

- Default chunk frequencies prevent complex chunks from dominating the output.

- The boredom value helps ensure variety in chunk selection.

- The precision value is an example of a level design choice encoded in the chunk selection policy.
The weight for the first match might be:

\[ w = 0.75 \times 0.7^0 \times 1 = 0.75 \]
The weight for the second match might be:

\[ w = 1 \times 0.7^0 \times 1 = 1 \]
The weight for the third match might be:

\[ w = 0.5 \times 0.7^0 \times 0.2 = 0.1 \]
Chunk Integration

- Removes any overlapping components from the incoming chunk.
- Adds remaining components to the level under construction.
- This step could be used to enforce some global constraints.
The result of integration, assuming the second match is selected:
Post-Processing

- Specifies and expands terrain sprites.
- Implements global constraints on the distribution of enemies and powerups by removing some.
- Tries to patch up sprite inconsistencies.
Example

The level after post-processing:
The Chunk Library

- A total of 42 chunks.
- Ranges from 3x2 to 10x10 tiles in size.
- Hand-crafted chunks, some with authored complexity.
Example Chunks
Example Chunks
Example Chunks
Example Chunks
Summary

- Using human-authored chunks, ORE assembles a level by adding chunks one-at-a-time.
- The main constraint imposed is that added chunks are anchored via potential positions.
- The algorithm is highly customizable, and higher-level constraints can be imposed on it.
Failures
Failures
Future Work

- On-line generation for dynamic difficulty adjustment.
- An interface for mixed-initiative design.
- Automatic chunk library extraction.
- Application to other domains.