# The Technology of Jak & Daxter

Presented By Stephen White

Programming Director

Naughty Dog, Inc.

Naughty Dog is a Wholly Owned Subsidiary of Sony Computer Entertainment America, Inc.

# Jak & Daxter: The Precursor Legacy

- Published by Sony Computer Entertainment for the PS2 in December 2001
- 35 full-time developers
- 3 years to develop
  - 1 year of initial development
  - 2 years of full production

# Commercial Tools

- Operating systems used:  Windows NT, Windows 2000, & Linux.
- Software used:  Allegro Common Lisp, Visual C++, GNU C++, Maya, Photoshop, X Emacs, Visual SlickEdit, tcsh, Exceed, & CVS.

# Internal Tool Development

- The importance of tools.
- Large amount of data that needs to be processed.
- Data formats and data streams need to be able to be changed fairly easily.
- Poor planning can result in a large loss of productivity.

# The DB

- Boxed (tagged) data – could identify itself.
- The basic DB class was a type of generic container that could hold any number of other DB classes.
- Reading, writing, inspecting, copying, iterating, adding, and deleting can be done generically.
- New data can be added easily.
- Data is easily passed thru the pipeline.
- Any data file could be inspected by our generic db inspection tool.

# Script Driven Tools

- Most of our tools were script driven.
- Generic script parser:
  - Could parse C syntax.
  - Had some additional macro support.
- Easy to add new features and options to the tools.
- Not the best interface for artists.
- Caused some undesired build dependencies.

# From Maya To DB

- Only two stand-alone tools linked with Maya:
  - mbo (Maya Build Object)
  - mbl (Maya Build Level)
- Both tools did a fairly straightforward conversion from a Maya scene to our DB data representation.
- The other stand-alone tools used our DB data (not Maya's data representation).

# From Maya To DB (cont)

- This insulated the other tools from changes in Maya, peculiarities of Maya, and having to link with Maya’s libraries.
- Also allowed us to slightly cook and streamline the data to just what we were interested in.
- Kept data accessing nice and consistent for the other tools.

# Additional Tools in the Pipeline

- buildactor – processed animations and foreground geometry, and outputted run-time data.
- stripdb – stripped background geometry.
- visdb – precomputed visibility to aid in run-time occlusion culling.
- buildlevel – final processing of level data, and outputted run-time data.

# Additional Tools in the Pipeline (cont)

- tcomp – compiled our texture pages.
- texinfo – Windows tool used by artists to manage textures and texture pages.
- Various Maya plug-ins and MEL scripts to aid in modeling, lighting, texturing, instancing, creating/editing tfaces, placing shrubs, etc..
- Various other tools – Viewing db files, examining databases, processing sounds, debugging, etc..

# Tool Things We Did Wrong

- Didn't use a PC farm to process levels and actor animations.
- Overburdened our network.
- Poorly constructed data dependencies.
  - Small changes could cause needless and lengthy reprocessing of data.
  - Relied too much on file time/date stamp checking, which turned out to be extremely slow.

# Tool Things We Did Wrong (cont)

- Underestimated the difficulty of managing the audio, and had insufficient audio tool support.
- Processing FMA sequences was difficult, time-consuming, and error prone.

# Background vs. Foreground

- We typically refer to things that are mainly static as “background” (i.e., ground, cliffs, rocks, trees).
- We typically refer to things that can move around as “foreground” (i.e., an enemy).
- This is a seemingly arbitrary distinction, but affected how our data and renderers were constructed.

# Background Elements

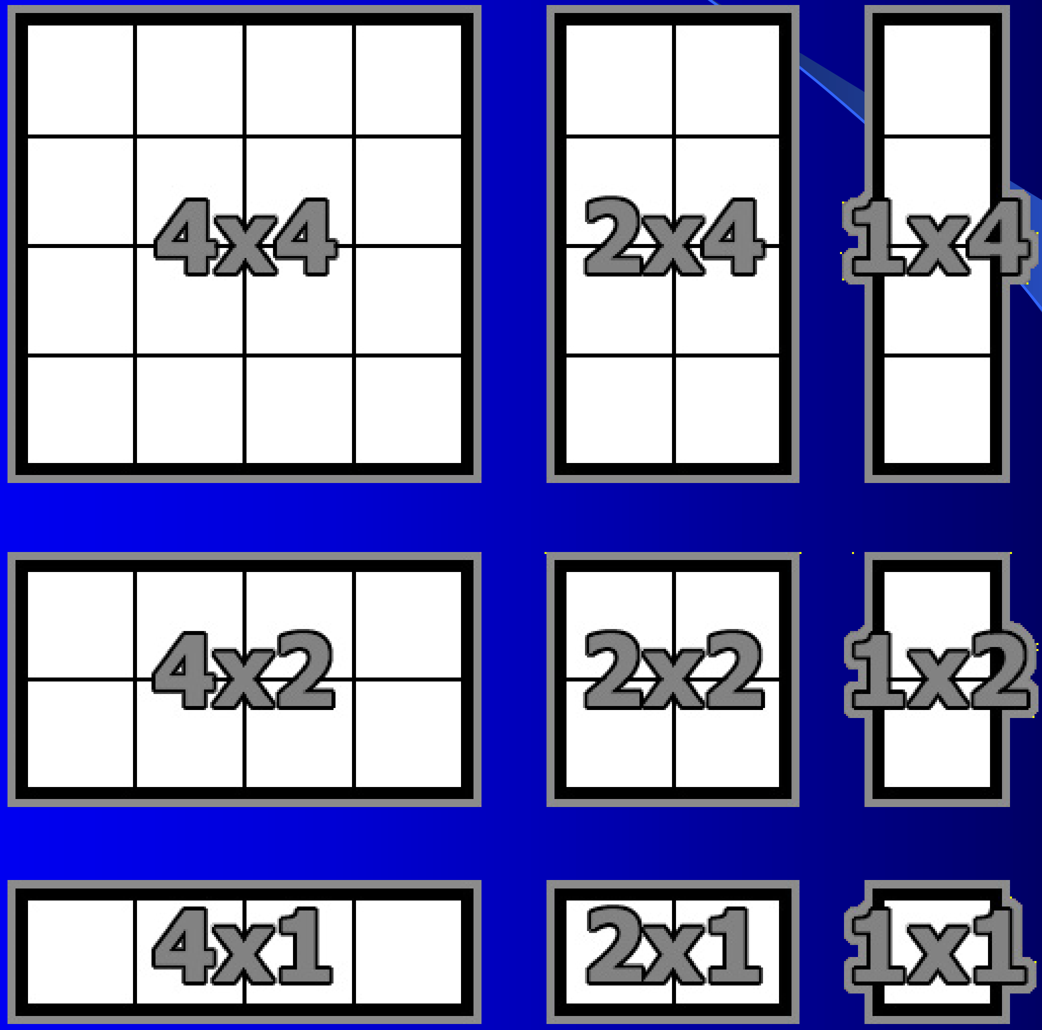
# Tface (Tessellating Face)

- Dynamically tessellates based on the distance to the camera.
- High detail up close, low detail far away.
- Composed of two primitive types:
  - tface quads
  - tface tris

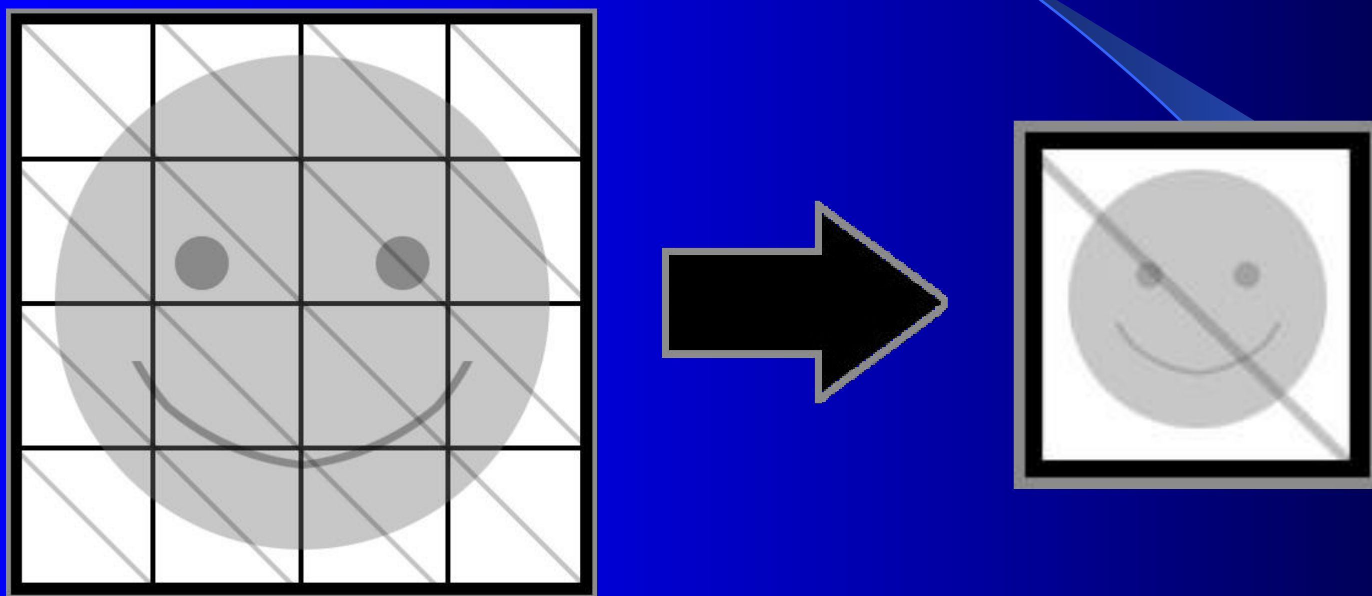
# Tface Quads

- Essentially a grid of polygonal quads.
- 1, 2, or 4 quads wide by 1, 2, or 4 quads tall (i.e., 1x4, 2x2, 4x2, etc).
- Modeled as a collection of quads in Maya.
- Custom Maya plugins allowed tface quads to be “constructed” from polygonal quads.
- Custom Maya plugins aided in setting/editing uv and corner color values.

# Tface Quad Examples



# Tface Quad Tessellation



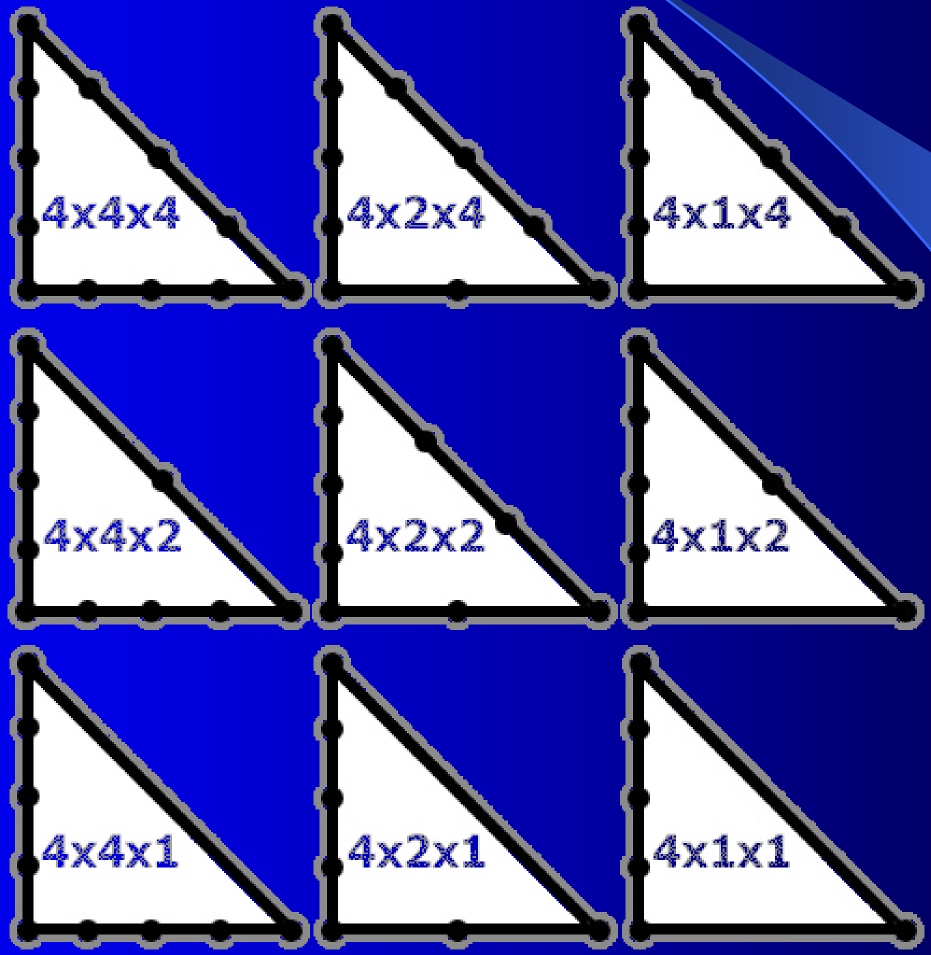
4x4 (16 quads) reduces to a single quad.

93.75% reduction in detail!

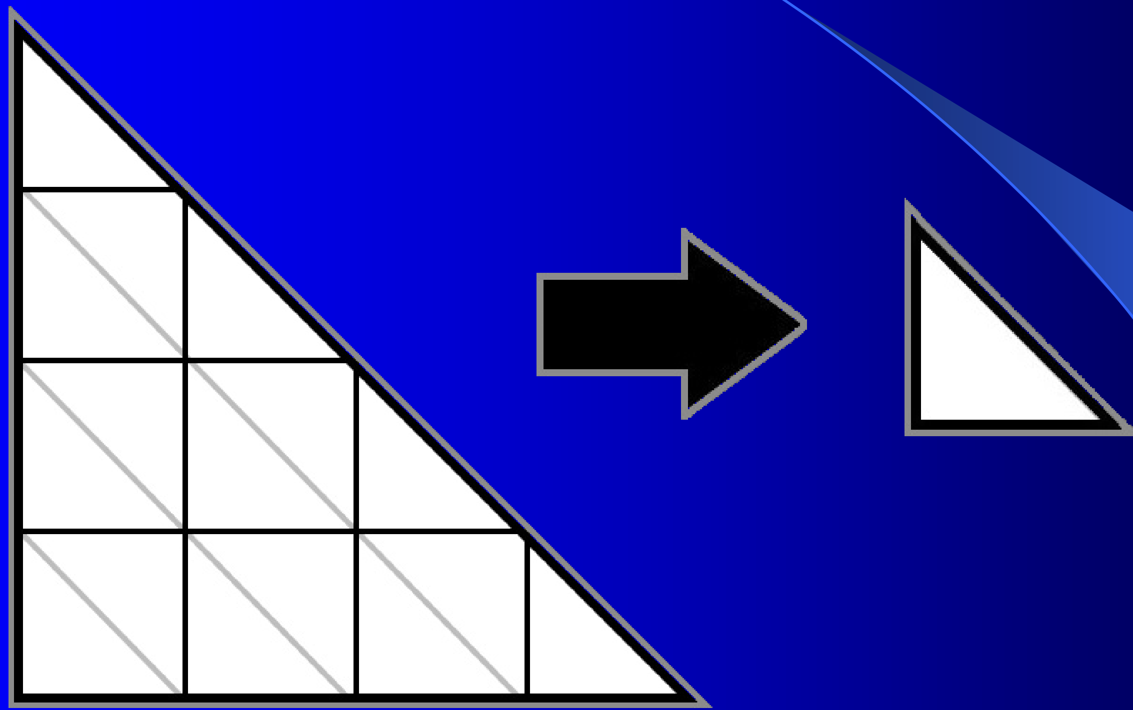
# Tface Tris

- 1, 2, or 4 subdivisions for each edge (i.e., 1x4x2, 2x2x4, 4x1x4, etc).
- Modeled initially in Maya as a single triangle, then tessellated by a plug-in into the required configuration.
- Custom Maya plugins aided in setting/editing uv and corner color values.

# Some Tface Tri Examples



# Tface Tri Tessellation



4x4x4 (16 tris) reduces to a single tri.

93.75% reduction in detail!

# How Lawyers Become Rich

- Sony is in the process of patenting how exactly we did the tessellation of our tfaces.

# Tfrag (Tessellating Fragment)

- Composed of a collection of tface tris & tface quads.
- Static, non-instanced (unique) background.
- The tfaces of a level were broken up by the tools into tfrags based on stripping and Vector Unit memory size constraints.
- Tfrag geometry indexed a 1024 color palette for corner color lighting.

# Tfrag Renderer(s)

- Used multiple renderers based on how a tfrag was to be tessellated (distance to camera).
  - Optimized renderers for various cases.
- Supported effects:
  - Opaque
  - Translucent
- Translucent tfrag was not sorted, so draw order could be wrong.
  - In practice, this wasn’t as bad as you’d think.

# Tfrag Collision

- Modeled as tfaces for convenience, but converted by the tools to tris.
- The artists could use either the geometry to render as the collision geometry or could build custom collision geometry as appropriate.
- Only one collision model was modeled for all tfrag, but the tools broke up the data into smaller pieces written out as a spatially subdivided tree.
  - Each node of the tree had a bounds sphere and up to 8 child nodes.

# TIE (Tfrag Instance Engine)

- Instanced:  one model (the prototype) drawn multiple times, using a different matrix and lighting per instance.
- Prototype modeled as tface tris & quads.
- Placed in a Maya scene file using ref nodes.
  - Ref nodes could be viewed expanded or collapsed using custom Maya plug-ins.
- Each instance was lit individually, but was then mapped to a single 128 color palette per prototype.

# TIE Renderer(s)

- Essentially instanced tfrag, but uses specialized renderers.
- Supported effects:
  - Opaque
  - Translucent
  - Environment mapped (thru the Generic Renderer)

# TIE Collision

- Modeled identically to tfrag collision, but each TIE prototype had its own collision mesh (or no collision at all).

# Shrubs

- Used for more than just plants.
- Fast renderer for rendering small details.
  - Data format closely matched what GS used to draw.
- Instanced (one model drawn multiple times).
- Prototype modeled as tris and quads.
- Placed in a Maya scene file using ref nodes.
  - A special Maya plug-in could also be used to "randomly" populate an area of ground with shrubs.
- Each instance had a color multiplier used for lighting.

# Shrubs (cont)

- Supported effects:
  - Opaque
  - Translucent
- Could become a single quad “poster” in the distance (far level of detail).
- Becomes translucent and fades away in the distance.
- An instance could sway in the breeze using a manipulation of its transformation matrix.

# Background Lighting

- The background was pre-lit up to 8 times using Maya.
  - An artist would light the level, then save the lighting information into a level color database.
  - The 8 sets of colors were typically used to achieve our “time of day” lighting.
  - At run-time the sets of colors could be cross-faded to create the appropriate lighting for the current time of day.

# Background Lighting (cont)

- Tfrag vertices indexed a 1024 color palette that was generated using 8 tfrag time of day palettes.
- TIE vertices indexed a 128 color palette that was generated using the prototype's 8 time of day palettes.
- Other effects, such as flickering lights, were also done using the 8 palettes.

# Misc Background Renderers

- Sky renderer.
  - Looking at the inside of a pyramid.
- Ocean renderer.
  - A giant tessellating grid.
  - Nearby polys were translucent, while farther polys were opaque.
  - Environment mapped.

# Actor Placement

- Placed using specially named locator nodes.
- Assigned a unique name (i.e., grunt-67) based on Maya DAG path and scene file.
- Assigned attributes in a hand-typed  level config script.

# Curves

- Linear and second degree curves.
- Parented under an actor node, and saved as part of that actor’s instance data.

# Nav Meshes

- Used for navigating enemies around our complex 3D world.
- Modeled as a relatively 2D mesh composed of tris.
- Described in detail in <u>Game Programming Gems 3</u>, published by Charles River Media, Edited by Dante Treglia.

# Volumes

- Modeled as polygonal meshes in Maya.
- Used for a variety of things, including visibility, water detection, etc.

# Foreground Elements

# Merc

- Used for enemies, platforms, FMA, etc.
- Modeled as polygonal meshes composed as tris or quads, and bound (weighted) to joints.
- Mesh deformed by joints using a max of 3 joint influences per vertex.
- Optionally vertices could also be deformed using blend shapes.

# Merc (cont)

- Used separate models for different levels of detail.  Auto switched between models based on the distance to the camera.
- Becomes translucent in the distance and fades away.
- Also used a special eye renderer to composite animating lids, pupils, and irises.

# Merc Blend Shapes

- Used mainly to do the highly animated facial expressions of the FMA characters.
- Multiple “blend targets” were created by the artists in order to generate various facial expressions.
- The animators could blend in the effects of multiple blend targets as desired.
- The artists used Maya’s blend shape support to both model and animate the blend shapes.

# Merc Renderer(s)

- Supported effects:
  - Opaque
  - Limited, unsorted translucency
  - Environment mapping (thru the Generic Renderer)
  - UV scrolling
  - Ripple (multiple sin waves thru a grid to make waves of water)
  - Vertex deformation using joints.
  - Vertex deformation using blend shapes.

# Foreground Lighting

- 3 dynamic lights used per merc object.
- Global “rgba multiplier” and “rgba add” used per object.
- Optionally, foreground could use time of day directional lighting information.

# Foreground Collision

- Multiple collision meshes allowed per actor model.
- Each collision mesh was modeled as a polygonal mesh using tris or quads.
- Each collision mesh was transformed at run-time using a single joint (no vertex deformation).
- Optional collision spheres bound to individual joints we’re also used, and were configured in code.

# Shadows

- A simplified model was used to cast shadows.
- Mesh deformed by joints using a max of 2 joints influences per vertex.

# Shadow Renderer

- A shadow volume was dynamically constructed at run-time based on extruding the transformed mesh, typically away from a light source.
- Once the extruded run-time volume was created, the shadow renderer using the classic shadow volume approach to create an alpha mask that was then used to darken the underlying pixels on the screen.

# Misc Foreground Renderers

- Sprite (particle) renderer:
  - 2D sprites
  - 3D sprites
  - Distortion sprites (simplistic refraction effect)
    - Heat effects
    - Blurs.

# The Generic Renderer

- Due to our large number of highly specialized renderers, certain effects were impractical to implement in each renderer.
- Other renderers (either foreground or background) could output to the Generic Renderer’s format in order to do specialized effects or to handle computational or memory-wise expensive cases.
- Some examples:
  - Environment mapping
  - Scissoring (triangle subdivision)

# The Generic Renderer (cont)

- The generic renderer was slower than the specialized renderers, but was much easier to extend to do various effects.

# Enough About Foreground and Background

# Visibility (Occlusion Culling)

- Visibility information about which fragments and actors could be seen from a given location was precomputed by the tools.
- A BSP tree was used to look up the appropriate visibility information at run-time based on the position of the camera.
- The visibility information was essentially a string of bits specifying “draw” or “don’t draw” for the various elements in the scene (I.e., tfrags, TIEs, shrubs, actor, etc).

# Visibility (cont)

- The bit strings were stored compressed, and were decompressed as needed at run-time.

# Defining Where an Actor was Visible

- Actor visibility was specified using a bounding box around everywhere that the actor could reasonably travel.
  - The actor bounding boxes could be auto-generated at run-time using both the actor nav meshes and by playing the game for awhile.
- An actor’s visibility could be set specifically in the level config script for given locations.

# Spooling Level Data

- Used two 10M data level buffers.
- A single game level was composed of one or more data "levels".
- We used careful planning and layout to insure that no more than two data "levels" were visible at any one time:
  - The level the player was currently in.
  - The level the player was heading towards.
- As the player approached a new level, it was loaded into the other level buffer.

# Spooling Level Data (cont)

- If the player changed his mind and headed towards a different level, then the level that was being loaded was abandoned, and a new level load was started.
- Used “load boundaries” to determine when to start loading or when to display a new data level.
- Used low resolution models to represent portions of other levels seen in the distance.

# The Spooled Level Data

- The level data contained:
  - Background (geometry, collision, textures, visibility, etc).
  - Foreground (geometry, collision, textures, animations, etc).
  - Sound.
  - Level specific code.

# Other Spooled Data (Not Part of the Level Data)

- Certain lengthy individual animations (i.e., Jak's bored/idle animations).
- FMA sequences:  animation for multiple characters, and audio.
- Spooled audio for lengthy sound effects.

# G.O.A.L.
# (Game Object Assembly Lisp)

# What is GOAL?

- GOAL is our custom compiler based on Lisp (well, actually Scheme).
- Practically all of the run-time code (approximately half a million lines of code) was written in GOAL.
- Only the IOP code and a small amount of kernel code was written in C.

# GOAL Features

- Object-oriented language.
- Extremely simple syntax.
- Powerful macro capability, far superior to C's preprocessor or C++'s templates.
- Listener
  - Code could be executed live at a "listener".
  - Code could be compiled, downloaded, and linked without interrupting gameplay.
  - Data structures could be inspected or modified live.
  - Rapid tuning and debugging.

# Goal Features (cont)

- Non-preemptive, cooperative multi-tasking.
- The “suspend” operation.
- Stack variables were preserved across a suspend.
- Code could be written to flow naturally, rather than the typical C++ AI behavior schemes which use state based “update” callbacks that have to deduce what to do next.
- Extensive support for AI scripting.

# Goal Features (cont 2)

- Unified set of assembly op-codes consistent across all five processors of the PS2.
- Seamless intermixing of high-level code with low-level (assembler) instructions.
- Register coloring when using assembler instructions.
- Code could be loaded as part of the level data.

# Difficulties with GOAL

- Compiler took over a year to write, and was fully developed and supported by one person.
- We suffered with a buggy, and quirky compiler as GOAL was being developed.
- An unfamiliar language for programmers to use.
- Difficult learning curve.

# Difficulties with GOAL (cont)

- Isolated from other developers. Couldn't use other people's tools or libraries.
- The compiler would periodically run out of memory, and require several minutes before it could compile again.

# Any Questions?

swhite@naughtydog.com