Brian Moriarty | Lectures & Presentations | Lehr und Kunst mit Perlenspiel

Lehr und Kunst mit Perlenspiel (2012)

Given 5 March 2012 as the keynote lecture of the Education Summit at the 2012 Game Developer's Conference in San Francisco.

The original presentation featured several samples of student work. Some of these are available at the Perlenspiel web site.

§

The title of this lecture is "Lehr und Kunst mit Perlenspiel."

Lehr und Kunst happens to be the German motto of Worcester Polytechnic Institute, where I am a Professor of Practice in the Interactive Media and Game Development program.

Lehr shares the root of the English word *learning*. WPI officially translates this word as *theory*.

Kunst is translated as *practice,* as in craft or art.

That, I promise, will be the only time the word “art” appears in this lecture.

§

I remember my first semester as a senior in high school.

It was time to start thinking about college. Time to decide where to go and what to study.

I had no idea what I wanted to do with my life.

I liked to read, draw, paint and make movies when I could afford to.

I liked computers, too, though I’d never actually used one, except for the toy computers I’d built myself.

But my chief passion, then as now, was music.

I remember going to the guidance counselor’s office and bringing home catalogs from the music academies in the Boston area.

The Longy School, the Berklee School, the Boston Conservatory, the music departments of the big universities.

I found myself drawn to NEC, the New England Conservatory of Music, turning the textured pages of their expensively printed catalog and reading the course descriptions.

In a daydream, I imagined myself settling into a rigorous, time-tested curriculum based on centuries of tradition, guided by world-renowned musicians and composers.

Mastering the intricacies of harmony and counterpoint, navigating the late sonatas of Beethoven and Schubert, wearing a tuxedo and performing in recitals right around the corner from Symphony Hall.

But when I turned to the page of admission requirements, my heart sank.

No reputable academy of music admits anyone without an audition.

And although I had some minor ability as a self-taught pianist, I could barely read a note.

The page on tuition was equally exclusive.

I ended up going to a cheap state university to study visual design, based on a portfolio of hand-painted 16mm films, and somehow graduated with a degree in English.

Forty years later, after a long and curious career, I find myself on the far side of the college fantasy.

As I speak, high school seniors are turning catalog pages, fantasizing about what it will be like to become a student of digital game design.

What are our courses like in their daydreams?

No matter what they're imagining, one thing is certain.

It won't be like going to a music academy.

There won't be any auditions. Little or no prior experience is expected at any game school I'm aware of.

They won't have to sight-read anything, either. We have no way of notating game mechanics. A lot of us probably couldn't agree on what a "game mechanic" is anyway!

They won't face a rigorous, time-tested curriculum, either. I know, because I looked for one.

Upon being appointed to WPI, the first thing I did was try to figure out how other people were teaching digital game design.

I discovered what most of you here know all too well.

There are no standard texts, tools or curriculum.

We don’t have any books like *Gradus ad Parnassum,* published in 1725 by Johan Fux, which synthesized the musical pedagogy of three centuries into a step-by-step method for teaching counterpoint and fugue.

The *Gradus* served as a textbook for Bach in the eighteenth century, Haydn and Beethoven in the nineteenth, Hindemith and Strauss in the twentieth.

The techniques it pioneered are still in wide use today.

We also lack a common engine, or instrument if you will, that can be used to express, practice and compose games.

We have plenty of keyboards, but no piano.

This fancy of mine, this daydream of game design being taught with the austere refinement of fine music may strike some of you as grandiose.

Maybe even a little bit elitist.

But the idea does have an elite precedent.

In 1943, as the Second World War was approaching its climax in Europe, an elderly German living in Switzerland published the final novel of his career.

This two-volume masterpiece was later cited by the committee that awarded him the Nobel Prize in Literature.

The author was Herman Hesse, and the novel was *Das Glasperlenspiel, The Glass Bead Game*.

Set in a more peaceful Europe hundreds of years in the future, The Glass Bead Game describes an elite academy devoted to the design and study of the ultimate intellectual challenge, a kind of super-Scrabble combining music, mathematics, linguistics, meditation and even calligraphy in a competitive ritual requiring a lifetime to master.

The rules of the Glass Bead Game are never fully explained. Hesse's descriptions are elliptical, tantalizing, and have fascinated generations of readers.

A quick Web search reveals numerous attempts to design a playable game based on Hesse's outline. Musicians and artists have also been inspired by the game's encyclopedic scope and formal elegance.

At least one minor religion appears to have been founded on its principles.

Hesse's vision of game design as a noble discipline has long been an inspiration to me.

But this vision was of little help back in 2009, when I was a given a few weeks to develop a course in digital game design.

Prior to this appointment, I had spent 27 years in the game industry designing an eclectic series of titles ranging from text and graphic adventures to full-motion-video rail shooters to smutty phone games, culminating in a talking teenaged Dora doll that was famously condemned for being too thin.

I never stopped to think very much about how I did what I was doing.

I just did it as well as I could, trying not to embarrass myself, and hoping to do better next time.

But now I had to come up with a lesson plan, assignments, and tests, and a rubric for grading. I had to pass myself off as a real Professor.

So I studied the courses offered by other schools. I read all the major textbooks on the subject, such as they are.

I found a few things I really liked.

This first book uses the methods and materials of paper gaming to introduce concepts essential for digital gaming. It offers a direct, hands-on approach that I find very appealing.

The second title navigates the opportunities and pitfalls of design with particular clarity.

And the third book emphasizes the creative process itself, considering not just *how* to build games, but *why*.

I also liked the idea of a *piecepack,* such as this one sold by Mesomorph Games. A piecepack is a collection of generic board game components, dice, markers and such. It's the closest thing I could find to a game composition toolkit.

I considered using a piece pack to teach my course. But there were two problems.

First, WPI already has a excellent course on tabletop game design.

One of our physics professors, Dr. George Phillies, also happens to be an authority on classic war boardgaming. He owns the largest personal collection in the world.

My second problem was more fundamental.

Please bear with me here. I am about to reveal one of my deepest prejudices.

Back in the early 80s, I entered the digital game business as technical editor of *ANALOG Computing,* a magazine devoted to Atari home computers.

I reviewed programming languages and hardware while writing utility programs, demos and games. My source code was printed in the magazine for hobbyists to type in by hand.

I was hired by Infocom to write 6502 assembly-language interpreters for their adventure games. Later, as an Implementor, I spent years writing tens of thousands of lines of code in ZIL, a domain-specific language based on MDL and Lisp.

I personally undertook virtually all of the SCUMM coding for Lucasfilm's *Loom*.

Although I am known chiefly as a designer, I am fluent in several programming languages, and have made my living *coding* most of the games that bear my name.

To me, digital games are *made of code*.

I've hired a number of game designers over the years.

If you put two designers with otherwise equal talent and credentials in front of me, I will always choose the one who knows how to write code.

I decided that the fundamental activity of my students ought to be, must be, *expressing game ideas in code*.

Now I had to choose an engine.

I looked at AGS, Flash, Flixel, GameMaker, Inform, Kodu, Love, PyGame, Scratch, Source, Torque, UDK and Unity.

I looked especially hard at MIT's Processing, which has many admirable qualities.

But none of these seemed quite right.

Some use proprietary scripting languages found nowhere else. Others are suited only for building particular types of games. Many have large APIs, fussy pipelines, or less than adequate documentation.

Our undergrad terms at WPI are only seven weeks long. I didn’t want my students spending most of the course getting up to speed.

I wanted them to build lots of games, not just one or two.

It gets worse. WPI has no course prerequisites. There was no guarantee that all of my students would be engineers. A third of the undergrads in our game program are artists with little or no coding experience.

Which brings up two more perennial problems: Graphics and sound.

I didn’t want some students doing all the coding while others wrangled assets and pipelines. We already have plenty of courses like that.

This was supposed to be a course on game *design,* not game *production*.

I wanted to give *everyone* a chance to express game ideas in code.

I wanted an engine simple enough for anyone to master in a couple of weeks, but deep enough to realize significant designs.

An engine using a well-documented, industry-standard scripting language that was inherently worth learning.

A language supported by a professional development environment like Eclipse, with breakpoints and debugging and profiling.

A language that would look good on a resume.

But I also needed an engine requiring no assets and no pipeline.

An engine so transparent that students could sit down and start doing useful work after just a few evenings of study, fingering out ideas like notes on a piano.

What I wanted was a *gameclavier*.

Alas. There are no Steinways or Bosendorfers in this business.

If I wanted a gameclavier, I was going to have to build it myself.

This is the point in the lecture where I'm supposed to press the space bar and triumphantly unveil my (quoting the program here) "sinister method for hurling students into the crucible of game design" with "an evil elegance worthy of the name Professor Moriarty."

(I may or may not know anything about teaching game design, but I do know a thing or two about writing lecture proposals.)

Let me tell you now about the education of Professor Moriarty.

My thinking in those crazy weeks before my first class went something like this.

Composers select and arrange notes. Painters mix pigments. Sculptors mold clay.

What is the working substance of digital games? In what medium are they generally realized? What is our canvas?

The obvious answer is: a screen.

Okay. What is a screen made of?

Regardless of hardware or genre, virtually all contemporary games are ultimately realized in a 2-dimensional raster of colored squares.

Modern operating systems and game engines have all but eliminated the need to address the individual pixels in a display.

But for us old-timers who cut our teeth in the late '70s and early '80s, changing the color of individual pixels was what making games was all about.

I submit to you, therefore, that a deep understanding of digital games and the tools used to create them requires a working familiarity with pixels.

A modern high-definition monitor incorporates more than two million pixels. That seemed a bit unwieldy.

32 x 32 pixel raster

So I decided to limit the raster of my gameclavier to a much, much smaller scale: just 32 x 32 pixels, the size of a medium Windows icon.

It occurred to me that the giant pixels in their grid looked like the beads on an abacus, or the tiles in a mosaic.

Piet Mondrian: *Composition: Checkerboard, Light Colors* (1916)

I considered calling call my gameclavier Mondrian, after Piet Mondrian, the Dutch artist famous for his paintings of big colored squares.

Unfortunately, Mondrian.com, .net and .org were already taken.

Then I remembered Herman Hesse, and *The Glass Bead Game*.

Perlenspiel

The original version of Perlenspiel was a native Windows application, written in C++ and scripted with Lua.

The current version, 2.1, is based on HTML5. It runs with no installation on any compliant web browser, and is hosted entirely in the cloud.

The design of this engine is just as simple as it looks.

A Perlenspiel web page is divided into two areas.

In the center of the page is the grid, a rectangular raster of super-sized pixels, or solid-colored squares.

Each square in the grid is called a bead. Originally I was going to make them look like real glass beads, but I've rather come to like the European austerity of the squares.

There's also a text box above the grid called the status line. It can be used to communicate short messages to players: the title of your game, high scores, time limits or simple instructions.

By itself, Perlenspiel is genre-agnostic. It has no built-in support for any particular type of game or style of interactivity.

Its behavior is entirely controlled by Javascript.

Javascript happens to be the most widely-deployed programming language in the world. It controls the look and behavior of nearly all of the billions of pages on the World Wide Web.

Javascript is as real as computer languages get. It's exhaustively documented, and well supported by dozens of professional development tools, including Eclipse.

Javascript looks very good on a resume.

The Perlenspiel engine is itself written in Javascript.

When a player moves and clicks the mouse over the grid, scrolls the mouse wheel, or presses a key on the keyboard, the engine tells your script what the mouse and keyboard are doing.

Your script can respond by commanding the engine to change the appearance of one or more beads, play sounds, or display a message in the status line.

This event-driven interchange between the engine and script creates the entire game experience.

Perlenspiel event calls

- **PS.Init ()**
- **PS.Click (x, y, data)**
- **PS.Release (x, y, data)**
- **PS.Enter (x, y, data)**

- **PS.Leave (x, y, data)**
- **PS.KeyDown (key, shift, ctrl)**
- **PS.KeyUp (key, shift, ctrl)**
- **PS.Wheel (dir)**
- **PS.Tick ()**

All engine functions and variables share the prefix **PS**.

PS.Init() is called once, when a game begins. You use this call to establish the initial dimensions of your grid and perform any other setup you need to begin your game.

PS.Click() is called when a mouse button is pressed over a bead. The x and y parameters specify the column and row of the bead that was clicked. The data parameter will be explained in a moment.

PS.Release() is called when a mouse button is released over a bead.

PS.Enter() is called when the mouse cursor enters a bead. **PS.Leave()** is called when the cursor moves away from a bead.

PS.KeyDown() is called when a key is held down, and **PS.KeyUp()** when that key is released.

Finally, **PS.Wheel()** is called when the mouse wheel is scrolled forward or backward.

In addition to these player-generated events, Perlenspiel has a single clock that can be used to create timed events and animations.

You start the clock by calling **PS.Clock()** with the frequency you want, expressed in hundredths of a second. The script function PS.Tick is thereafter called at the specified frequency.

That’s it!

Composing a Perlenspiel game consists of nothing more than responding to these nine event calls with commands that change the appearance of the grid, the beads or the status line, or that play sounds.

There are only two commands for controlling the grid itself, both prefixed with **PS.Grid**.

PS.GridSize() changes the dimensions of the grid to anything from a single giant bead to a maximum of 32 x 32 beads. The grid can be resized anytime, even in the middle of a game.

You control the background color of the grid and its surrounding web page with **PS.GridBGColor()**.

Perlenspiel bead commands

- **PS.BeadColor (x, y, rgb)**
- **PS.BeadAlpha (x, y, alpha)**
- **PS.BeadBorderWidth (x, y, width)**

- **PS.BeadBorderColor (x, y, rgb)**
- **PS.BeadBorderAlpha (x, y, alpha)**
- **PS.BeadGlyph (x, y, glyph)**
- **PS.BeadGlyphColor (x, y, rgb)**
- **PS.BeadFlash (x, y, flag)**
- **PS.BeadFlashColor (x, y, rgb)**
- **PS.BeadData (x, y, data)**
- **PS.BeadAudio (x, y, audio, volume)**
- **PS.BeadFunction (x, y, exec)**
- **PS.BeadTouch (x, y)**
- **PS.BeadShow (x, y, flag)**

The commands for controlling the individual beads are a bit more extravagant. There are a grand total of 14, all prefixed with **PS.Bead**.

PS.BeadColor() sets a bead to any of 16 million hues.

PS.BeadAlpha() controls the alpha transparency of a bead against the background color of the grid.

Every bead has an optional solid border, up to 8 pixels wide. The three **PS.BeadBorder** commands let you control the width, color and transparency of the border on each bead individually.

Beads can contain a optional glyph or symbol, which can be any one of the million-plus UTF-8 characters. The pair of **PS.BeadGlyph** commands let you decide which glyph is displayed along with its color.

By default, beads “flash” momentarily when their color is changed. The **PS.BeadFlash** commands let you turn this flash effect on or off, and control the flash color.

Each bead can be associated with an arbitrary Javascript value – any number, string, array, object or function. **PS.BeadData()** lets you assign this value to a bead, which then appears in the data parameter of the event calls I showed you earlier. This feature makes it easy to create classes of beads.

Beads can have a sound attached, which is played when the bead is clicked, and also a user-definable function, which is called when the bead is clicked.

You can simulate the effect of clicking on a bead without the player actually touching it by calling **PS.BeadTouch()**.

Finally, you can make beads active or inactive with **PS.BeadShow()**.

Each of the bead commands begins with two parameters, x and y, which specify the zero-based row and column of the bead you want to change.

If you use the constant **PS.ALL** in either parameter, all of the beads in the specified row or column are affected. If you use PS.ALL in both parameters, every bead on the grid is affected.

PS.StatusText() and **PS.StatusColor()** change the text and color of the status line.

By default, status text fades in when it is changed. You can turn this effect off or on with **PS.StatusFade()**.

Perlenspiel is equipped with a cloud-based library of about 400 sound effects and musical notes, ranging from generic clicks and pops to a full harpsichord, a xylophone and an 88-note piano.

These four commands let you load, play, pause and stop these sounds. All of the sounds are available for instant preview on the Perlenspiel web site.

Finally, there's a debugging window under the grid that you can open to display in-game diagnostics, along with a handful of utilities for calculating color values, generating random numbers, and extracting pixel data from image files.

This setter is also a getter.

In Perlenspiel, the setters are also the getters. This means that the commands return values that represent the current state of the related grid element.

For instance, if you call PS.BeadColor() with or without a color parameter, it returns the color of the bead at the specifed location.

All of the other Perlenspiel commands work the same way.

§

Since fall of 2009, I've taught four sections of Digital Game Design I at WPI.

Over this period, my students composed over 250 toys, gizmos and games on the Perlenspiel gameclavier.

The seven-week curriculum consisted of fourteen two-hour classes.

In Class 1, I explained the structure and goals of the course, introduced Hesse's *Das Glasperlenspiel,* and asked the students to read the novel's 50-page introduction in hopes that they, too, would be inspired by its vision.

This assignment was always unpopular. Game undergrads don't like to read. I no longer assign this reading.

I next held up a fine Moleskine grid-ruled notebook and informed the students that they would be required to maintain a creative journal as they worked on their projects, and submit it at the end of the course for grading.

This assignment was also unpopular. Game undergrads don't like to write. I still assign the journal, but I don't bother grading it anymore.

I finally introduced the Perlenspiel gameclavier, with its featureless raster of giant pixels, and announced that they would be using it to complete six projects in the following seven weeks: one toy, one puzzle and four games.

Student reactions ranged from bewilderment to relief to delight.

The bewildered students were the most numerous.

They thought they were going to be spending the course inventing cool characters, writing dark backstories involving amnesia or the apocalypse, devising novel variations on chain saws and nail guns, and animating 3D zombies, robots and vampires.

The relieved students thought they were going to be spending the course brainstorming with the bewildered students. They thought they'd be writing spreadsheets and design documents, and never getting to actually build much of anything.

The delighted students were the ones who were making weird little games in their spare time anyway.

Class 1 also included a lecture with some basic definitions for *play, toy, game* and *puzzle*.

My nested definitions are inspired by *The Adventures of Tom Sawyer,* in which Mark Twain memorably remarks that "Work is whatever a body is obliged to do, and play is whatever a body is *not* obliged to do."

I then presented a brief summary of how the Perlenspiel engine works, similar to the one I just gave you.

I pointed everyone to a web site where they could read about Perlenspiel, study a few simple code examples, and download the files needed to use it.

I gave them the URLs of ebooks and Web sites describing Javascript.

Finally, I asked every member of the class, engineers and artists alike, to compose a toy or gizmo on Perlenspiel, due in the very next class, three or four days later.

I offered no instruction in programming or Javascript.

I did not show them any of the projects created by previous classes.

But I did give them permission to help one another, providing that due credit was given.

We spent most of Class 2 looking at and discussing the toys that everyone had built.

I then split the class into project teams of two students, with an art student in every team when possible.

The first team assignment was to prototype a puzzle with Perlenspiel.

These were the project requirements:

- It had to meet my definition of a puzzle (a game that can be solved).
- It had to be designed for a single player.
- It had to be replayable by someone who had already finished it, and still be fun. This means that the activity (physical and/or mental) required to solve the puzzle must be significantly different each time it is played.

- It had to be entirely self-documenting. No off-screen instructions or explanations should be necessary to use it.

A prototype of the puzzle was due in the next class. It didn’t have to be polished, but it did have to run well enough so other people could try it.

Class 3 was a playtesting studio. Teams paired up randomly to play each other’s puzzles for ten minutes, asking questions and taking notes. Then the teams swapped sides.

This was repeated three times over the course of the class, so that every team got to play three different puzzles, and get their game played by three different teams.

The teams were asked to use their testing notes to complete and polish their puzzles in time for Class 4, which was spent critiquing the completed puzzles.

The next four assignments were game projects, each occupying two class periods.

The first period was devoted to playtesting, the second to presenting and critiquing the games.

The first project in the series was a straightforward one-player game.

The second project was for a two-player game. Play could be either competive or cooperative, but had to work using the keyboard and/or mouse of a single computer.

By the time they'd completed the 2-player games, the teams had been working with Perlenspiel for just over a month. I decided it was time to start squeezing.

For the third game assignment, I gave them a menu of design restrictions.

All of the games had to conform to the following requirements:

- The game must have a title.
- The game must be for one player.
- Sound effects and/or music must be used.
- If a title screen is used, no background story or game instructions are permitted on it.
- It must be entirely self-documenting. No out-of-game instructions or explanations should be necessary to play it.

In addition to these general requirements, each team had to choose *any three* of the following five restrictions:

- The grid cannot exceed 16 beads in either dimension.
- No glyphs allowed.
- No timer allowed (that is, no calls to PS.Clock).
- No keyboard input allowed.
- No intelligible words in any language allowed

anywhere, except for the game title in the status line.

For the fourth and final game assignment, I turned the screws even harder.

- The grid cannot exceed 16 beads in either dimension.
- Must be for one player.
- No glyphs allowed.
- The game must use either mouse input or the four arrow keys and/or WASD for all user input (but not both).
- Sound effects and/or music must be used.
- It must be entirely self-documenting.
- The final submission must be functionally complete, stable and polished enough to add to your portfolio.

In addition to the above requirements, the gameplay — that is, the actions and consequences in the game, not just the presentation — had to exemplify one of the following abstract themes:

- Fate
- Transformation
- Recurrence
- Harmony
- Pyrrhic victory

§

Should you decide to use Perlenspiel your lab, there are certain things to watch out for.

Because the display is so restrictive, the better students will waste no time in pushing the edges to find out what they can get away with, both from the engine and from the professor.

In the first version of Perlenspiel, there was no hard limit on the dimensions of the grid or the size of the window. PS.Grid() would accept any values that resulted in a bead size of at least one screen pixel.

A few of the geekier types exploited this capability to produce conventional low-res games, similar to the ones we used to make for antique consoles and handhelds.

They were using the beads as *picture elements,* not as game elements.

In one extreme instance, a group of engineers harnessed Perlenspiel to recreate the appearance of a Gameboy. You could scroll around to look at the map of a Mario game, based on the original data.

You could even move Mario around the map.

Another thing to watch out for is cut scenes. Even with a 32 x 32 bead limit, some students will not be able resist temptation to setup their backstory with non-interactive animatics.

And sooner or later, a team of wise guys will use the bead glyphs to try their hand at a text adventure.

The most memorable of these was a riotously funny game called *You are British, What Will You Do?*

You are British turned out to be a wonderful teaching moment. It was the first time many of my students had ever seen a text adventure.

The class roared with laughter as we played it together! It earned one of the most positive reactions of any Perlenspiel project.

When we had finished our fish and chips, I pointed out that text adventures had once represented the state of the art in gaming, and that, for a time, it was even possible to make a living writing them.

I told them to remember *You are British* the next time they had something funny, strange or thoughtful they wanted to express in a game, but didn't think they had the resources to produce.

§

The Perlenspiel gameclavier and my curiculum for using it were experimental.

By some measures, it seemed to be a success.

The student course evaluations were unanimously positive. You all know how important that is.

Many students told me it was one of the best experiences of their academic career.

Nevertheless, looking back on the experiment as I wrote this lecture and assembled the video, I realized that I had not built the gameclavier I was aiming for.

I thought I was building an engine to teach digital game design.

What I actually did was build an engine for learning *how* to teach digital game design.

Last December, after my fourth section of the class was done and graded, and everyone had gone home for the holidays, I decided to start redesigning the course based on what I'd learned. I started with the final exam.

I had been concluding the course with a written test, in which I asked for the definitions of design terms I'd lectured about. Stuff like "meaningful play" and "affordances."

For fall 2012, I decided to replace the paper exam with a live, in-class playthrough of a short game the students had never seen before, and ask them to identify the elements of design that were exemplifed by the experience.

Obviously this game had to be written in Perlenspiel.

The only stuff I had built with the engine myself were tech demos. I'd never actually used it to compose a complete, polished game.

On Boxing Day, over a steaming cup of holiday cheer, a suitable idea presented itself, and I started coding. I figured I’d be done before New Year’s.

Nearly three weeks later, the game was sort-of finished.

I had forgotten how difficult it is it write for bare pixels without sprites, or z-planes, or scrolling, or an animation system, or any of the conveniences we’ve all come to take for granted.

For the first time, I realized just how much I had been asking when I told my students to write six games in seven weeks, and what they had gone through to satisfy me.

Perlenspiel demonstrated to this old Professor how hard students will work if they are playfully and firmly challenged.

Leonardo da Vinci once wrote, “Art lives from constraints, and dies from freedom.”

And Orson Welles said, “The enemy of art is the absence of limitations.”

(Yeah, I know. I said “art” again.)

Perlenspiel is open source. Everything you need to get started is available at www.perlenspiel.org.

. . .