Introducing ‘Code Resituation’: Applying the Concept of Minimal Intervention to the Conservation Treatment of Software-based Art

Deena Engel and Joanna Phillips

The Electronic Media Review, Volume Five: 2017–2018

ABSTRACT

This paper proposes a new treatment method for the conservation of software-based art that was developed as part of the research collaboration between the Guggenheim Conservation Department and the Department of Computer Science at New York University. The new treatment method, termed “code resituation” by
the authors, is designed to serve artworks for which code intervention is necessary to restore the artwork’s functionality. Traditional code migration, as practiced by computer programmers, includes the deletion and replacement of non-functional original code. Intended behaviors and discernible output of an artwork would be recreated by means of contemporary programming languages, aiming for the most elegant and efficient programming solutions currently available. This migration approach, the authors argue, has the potential to strip an artwork of some or all traces of the artist’s hand. The artist’s choice of programming language, artistic expression as seen through nuances in the source code and algorithmic detail, code annotations, and unrealized drafts can all be lost in code migration.

Code resituation, instead, aims to preserve the original artist’s code, or significant parts of it, while adding restoration code to reanimate defunct code to full functionality. With the development of this new treatment approach, the authors apply the established conservation principle of minimal intervention to the treatment of software-based art. The new method of code resituation was successfully tested on two artworks from the Guggenheim collection, which were treated in the course of the Guggenheim’s initiative “Conserving Computer-based Art” (CCBA).

KEYWORDS: code resituation, code migration, minimal intervention, software-based art, computer-based art, conservation ethics, time-based media conservation, media conservation, conserving computer-based art, CCBA

INTRODUCTION

A software-based artwork’s appearance and functional behaviors are determined by its underlying code. The proper, artist-intended execution of this code is often
dependent on a specific hardware and software environment. If these dependencies irretrievably fail owing to technological obsolescence or breakdown, the work itself may require conservation intervention on its code level. This is especially true in cases in which emulation or virtualization of the artwork’s environment does not result in satisfying execution of the work’s behaviors, such as speed, color, or movement. In these cases, code migration can be a powerful treatment option: the original code is translated into a different, contemporary technology or programming language to recreate the work’s behaviors precisely, based on the functional rules laid out in the original code.

From a conservation standpoint, however, code migration has to be regarded as a radically invasive treatment. Even if the migration takes place only in the restoration version of an artwork, leaving the original version untouched, the migrated version will replace the original in the context of an exhibition or other display of the work. From here on, the audience will experience only the migrated version. As the authors established in their previous research, original code—although mostly hidden from the viewer—can carry important meanings that are worthy of preservation (Engel and Phillips 2019). The artist’s (or delegate programmer’s) individual choice of technologies, programming languages, or coding style is evidence of an artist’s practice and expression; studying the original code and its annotations offers deep insight into an artwork’s genesis and history of restorations. These layers of meaning are easily eliminated when code is migrated, as programmers tasked to repair software are commonly not concerned with the preservation of original code. To prevent or minimize this loss of meanings when conducting a code migration, the authors identified a new treatment approach they termed, “code resituation,” which allows preservation or even reanimation of defunct original code, including its algorithms.

This new practice of code resituation supports a widely acknowledged
Introducing ‘Code Resituation’: Applying the Concept of Minimal Intervention Treatment of Software-based Art – Electronic Media Review

Introducing established conservation ethics and practice guidelines (AIC 1994) to the examination and treatment of software- and computer-based art has been a central goal of this cross-disciplinary research. Since 2014, Guggenheim conservation staff and New York University (NYU) computer scientists have been collaborating to cultivate the intersection between art conservation and computer science to develop best practices for the examination, documentation, and treatment of software- and computer-based artworks (Guggenheim Blogs 2016; Guggenheim Blogs 2017; Engel et al. 2018; Guggenheim Blogs 2018; Engel and Phillips 2019). Code resituation was developed and tested during the treatment of two case studies—both web artworks—from the Guggenheim collection, Shu Lea Cheang’s Brandon (1998–1999) and John F. Simon Jr.’s “Unfolding Object” (2002). Both works required code migration from their Java applets to other programming and markup languages because they were no longer supported in contemporary web browsers and less invasive strategies, such as emulation, did not prove to be a treatment option in both cases (Engel et al. 2018; Guggenheim Blogs 2018).

WHAT DOES MINIMAL INTERVENTION MEAN HERE?

Across the many specialties of cultural heritage conservation, the notion of minimal intervention is widely accepted and applied to guide conservation and restoration treatments of all kinds. “Its application means that, among all the methods and materials that can be used to address a condition problem, the best choice is the one that alters the object as little as possible” (Appelbaum 2007, 302). Respect for the original calls for limited interference, as every treatment of an object poses risks and can have negative side effects. For example, by introducing consolidants into the original structure of a painting, unforeseeable
long-term changes can be introduced. For software-based pieces, risks of invasive code treatments can include the introduction of unsustainable technologies or the misinterpretation of intended functional and aesthetic behaviors. Minimal intervention as a principle strives to support the removability of treatments and thereby the future retrievability of artworks. It intends to not only preserve the legibility of the original but also the traces of an artwork’s histories of production, use, display, appreciation, and repair.

In spite of its undoubted usefulness as a concept, the limits of minimal intervention have been critically discussed in conservation theory (Villers 2004; Muñoz Viñas 2009). Two main points of critique are the vagueness and subjectivity of the term “minimal” and the unjustified optimism that treatment reversibility is actually achievable even in the case of minimal interventions. In his article “Minimal Intervention Revisited,” Muñoz Viñas provocingly concludes that “the only truly minimal intervention would actually be leaving the sheet [of paper] alone. In a practical (but also precise) sense, minimal intervention means no consequential intervention at all” (Muñoz Viñas 2009, 49). He acknowledges that every intervention comes with a sacrifice and moves to interpret the term “minimal intervention” as “minimal meaning loss,” for it is “the decreased ability of the object to transmit some messages, whether social, ideological, aesthetic or scientific, that we really care about” (Muñoz Viñas 2009, 55). Of course, in the case of software- and computer-based art, “leaving the work alone” would likely not minimize the intervention; on the contrary, if a critical time window for intervention is missed, for example, while a certain decompiler software is still available, doing nothing can well mean to actively and irretrievably lose a work. Here, proactive and timely intervention offers a better chance of navigating and limiting the loss of meanings.

Minimal intervention informs treatment choices in both qualitative (methodological) and quantitative (treatment expansion) dimensions. It demands
that treatments be limited to the actual damage (where possible) and not extend to intact, undamaged areas of the artwork. For example, before the notion of minimal intervention guided treatment choices in paintings conservation, even small tears in the canvas were commonly treated by lining the entire painting with a second canvas and thereby exposing undamaged areas of the painting to adhesives, pressure, and possibly heat. While this measure stabilized the tear, it also commonly introduced a host of negative side effects, such as the accelerated aging of the original canvas, altered craquelure patterns, or flattening of impasto in the paint layer. Today, with minimal intervention in mind, the treatment of the same damage would focus only on the tear instead of the entire painting. In order to enable such local treatments, new methods, such as the thread-by-thread tear-mending technique (Heiber 1996), have been invented.

As in paintings conservation, damage in software-based art rarely affects the entire artwork at once but rather occurs locally, in parts of the underlying code. Rather than seeing a software-based work fail all at once, it is more likely that support for specific built-in functions or other aspects of a programming language is deprecated over time, causing only parts of the artwork to fail or malfunction. This means that, in many cases, the restoration of the work’s behaviors and functions requires treatment intervention only in parts of the original artist-created code. This finding has prompted the research team to investigate whether treatment methods could be developed that allow programmers and conservators to minimize the invasiveness of code migration by targeting only the damaged areas and not recoding the entire artwork. It was the goal of this research to identify an approach that would allow the preservation of significant parts of the original code, including its specific coding style and the artist’s original algorithms, even in cases in which the artwork had to be migrated from one programming language to another.
A PROGRAMMER’S APPROACH

Minimal intervention is an approach that is not intuitive to programmers. Professional source code development prioritizes efficiency and effectiveness over attention to authorship and authenticity (Engel and Phillips 2019). From a programmer’s perspective, code updates or changes are best handled in an automated way, that is, by writing a program or script that is designed to render the changes into the source code files that require revision. If code changes for a migration cannot be automated, a programmer might next evaluate the cost versus benefits of rewriting the code entirely, in which case the choice of the new technology or programming language would rest on an evaluation of the efficiency, accuracy, and cost of moving forward. Any code that is salvaged from an earlier version would be a result, not a goal, of the migration. Well-written production software is modularized and parameterized so that, ideally, updates to specific functions or other aspects of the software can be addressed individually without impact on the program as a whole. Artist–programmers, however, are not held to the same standards as commercial software programmers. Similarly, conservators base decisions on a different set of standards than programmers use for standard and commercial software maintenance. The expected result for both approaches is that all software is expected to run as designed to meet the goals and requirements that were initially stated, but the decision path in arriving at this point can be very different for works of software–based art in comparison with commercial software.

APPLYING MINIMAL INTERVENTION TO TWO CODE MIGRATIONS

In the course of this research, two legacy web artworks in the Guggenheim collection, 20 and 16 years of age, were treated to restore their functional and aesthetic behaviors. Both treatments required code migration from one technology or programming language to another because their original code
included Java applets and deprecated HTML tags that are no longer supported by contemporary web browsers. Code intervention was the preferred treatment method, as more conservative methods, such as emulation, had to be ruled out in these specific cases after extensive prototyping and testing failed to produce the desired results for a successful restoration. Detailed accounts of the treatments of both web artworks—Shu Lea Cheang’s *Brandon* (1998-1999) and John F. Simon Jr.’s *Unfolding Object* (2002)—are published elsewhere (Guggenheim Blogs 2017; Engel et al. 2018; Guggenheim Blogs 2018; Engel and Phillips 2019).

PRESERVING THE ORIGINAL CODING STYLE

Brandon, a sprawling, interactive web artwork with dozens of pages and pop-up windows (Guggenheim Collection Online, n.d., Cheang), featured so-called <blink> tags to make text flash on and off. Modern browsers no longer recognize the <blink> tag, and before the restoration of Brandon, blinking words across the site displayed as static text only.

In Brandon’s original HTML coding, all text between two blink tags was animated to blink; in the example featured in figure 3, the text Answer yes or no would blink.

```
13  <TD ALIGN= CENTER VALIGN= MIDDLE>
14  <FONT SIZE= "3"> <BLINK> <B> ANSWER YES OR NO </B> </BLINK></FONT>
15  <BR>
```

Fig. 3. Screenshot of Brandon’s original HTML coding, showing the <blink> tag that was used to make words blink. Courtesy of the Solomon R. Guggenheim Museum.

Without minimal intervention in mind, the treatment approach here would have been to deactivate this defunct original HTML coding and rewrite the blinking functionality entirely, for example, by using CSS style sheets or the programming language JavaScript. However, with minimal intervention in mind, the project programmer Emma Dickson kept the original HTML coding active and untouched and wrote a JavaScript function outside of the artist’s code that would tell the program to “go and find the old blink tags in the original HTML, and animate the text between two blink tags to make them blink again” (fig. 4).

This way, the original HTML coding could be retained and with it the artist’s coding style and expression. Other defunct elements in Brandon were treated in similar ways: wherever possible, the original code was retained and reanimated with additional external restoration code.
Fig. 4. Screenshot of Brandon’s restoration code: a JavaScript function written by the project programmer, Emma Dickson, searches for the deprecated <blink> tags in the original HTML and reanimates the text between them. Courtesy of the Solomon R. Guggenheim Museum.

PRESERVING ORIGINAL ALGORITHMS

The web artwork *Unfolding Object* allows users to unfold an “object” with mouse clicks and create their own artwork online (Guggenheim Collection Online, n.d., Simon). The lines inside the unfolded pages indicate the previous engagement of other users with these same unfolding pages. *Unfolding Object’s* visual design, colors, color gradation, the shapes of its pages, and the way that the pages unfold are all based on artist–created algorithms.

An algorithm is a set of functional steps that can be formally defined and thus repeated with the same results each time. To create software–based works, artists use programming languages to express their algorithms. In the history of mathematics, algorithms date back to the Ancient Greeks. Writing and studying algorithms is one of many creative processes in mathematics and computer science. The same can be said of visual artists working with software. Original algorithms can be regarded as evidence of an artist’s creative practice and expression that he or she applied to produce the design and functional behaviors.
of an artwork.

To illustrate how algorithms determine artwork behaviors, the management of *Unfolding Object*’s changing colors serves as a good example. Depending on the time of day, the object and its background are assigned different colors. When the piece is first accessed by a user (fig. 5), an algorithm (fig. 6) is triggered that identifies the time of the internal clock of a user’s computer and selects the appropriate color of the “object” as measured by hue. In another calculation, the background color is assigned by devising a near, but not exact, opposite or complementary color. Both of these colors are then rendered into RGB values for display.

![Algorithm snippet](image)

```java
float saturation = (float)1.0 - ((mTimeNow - mLastDate)/(float)(48-60+60-1000));
float hue = (float)(new Date().getHours())/(float)23.0;
theNewPage.mBaseHue = hue;
theNewPage.mColor = Color.getHSBColor(hue, saturation , (float).5);
theNewPage.mPath = "0";
theNewPage.mNumHits = 0;
theNewPage.pageCorners = new Corner[maxPoint3Dsi];
...

//set background color of applet
hue += .4375;
if(hue > 1.0)
  hue = 1.0;
mBackColor = Color.getHSBColor(hue, (float).75, (float)1.0);
this.setBackground(mBackColor);
```

Fig. 5. (left) The opening page of John F. Simon Jr.’s *Unfolding Object*, 2002, before interaction by the user. Courtesy of the Solomon R. Guggenheim Museum.

Fig. 6 (right) An excerpt in the Java source code that shows the algorithm that determines the color selections of John F. Simon Jr.’s *Unfolding Object*, 2002, when opened by a user. Courtesy of the Solomon R. Guggenheim Museum.

Given the significance of artists’ original algorithms, one treatment goal for *Unfolding Object*’s code migration was to preserve Simon’s original algorithms, even if parts of the artwork’s Java code had to be migrated to the browser-supported programming language JavaScript.
INTRODUCING CODE RESITUATION

A key principle that enables the preservation of original algorithms and other sections of original code in the course of code migration is the fact that an algorithm or other code excerpt can often run functionally unaltered in a variety of programming languages. This meant for the code treatment of Unfolding Object that instead of deleting or deactivating the original Java code and rewriting the artwork behaviors in JavaScript, it was possible to retain Unfolding Object’s original algorithms and other relevant code sections and integrate them into the restoration code. In other words, excerpts of the original Java code were resituated into the context of the new JavaScript restoration code. In turn, the new JavaScript code was tailored to support (and in parts to complement or compensate for) the functionality of the original code segments so that their calculations and functions remain active in the context of the new restoration code. The authors named this procedure code resituation.

To accomplish code resituation, the first step is to clone the artwork’s entire original source code. This clone becomes the restoration version of the piece: all code treatments occur in the restoration version only, leaving the original untouched. In the next step, the local damages that require treatment, defunct lines or blocks of code are identified. New code is inserted into the restoration version to translate defunct features from the original programming language so that the functionality of the destination programming language mimics the functionality of the original programming language. For example, in the case of Unfolding Object’s defunct Java applet, new JavaScript code was inserted so that arrays (a simple data structure containing multiple values) would behave as they do in Java. At the same time, original algorithms and other code sections remained unaltered where possible or only minimally altered where changes to the syntax were necessary to restore their functionality. The activation of the original code was supported by writing external JavaScript functions while
retaining the original naming conventions so that the new code would be consistent with the original with respect to variable, function, file, and other names and their intended execution. When practicing code resituation, it is important to clearly delineate the original code from the restoration code. This is done by annotating the code (Guggenheim Blogs 2017; Engel et al. 2018; Engel and Phillips 2019). Original artist comments were retained unchanged in the restoration source code and new comments in the code were added to identify the treatment interventions.

The choice of programming language can prove essential to accomplishing code resituation. In earlier research, criteria were developed to guide the decision-making on selecting a programming language for migrating a work of software- or computer-based art (Engel and Phillips 2019). Finding a newer version of the same programming language or environment should be prioritized over migrating the piece to an entirely different programming language. If migration to a different programming language cannot be avoided, a programming language should be selected that is similar in syntax and structure to the original. In both cases, consideration is needed to confirm the availability of analogous functions, libraries, and configuration options. From a programming perspective, the language selection must also support all of the systems and functional requirements needed for the specific artwork.

For Unfolding Object’s code resituation, JavaScript was selected as the restoration language, because it is sustainable, widely supported, well-suited to the web, and well documented. Moreover, the artwork’s original Java and JavaScript share many similarities in their structure and syntax. For example, both languages use curly braces to denote blocks of code, semicolons as statement terminators (to determine the end of a line of code) and double slashes for single-line comments. These similarities allowed the project programmer Karl Toby Rosenberg to avoid or minimize code changes when resituating segments of Unfolding Object’s
original Java into the context of JavaScript (figs. 7 and 8).

```javascript
theNewPage = new Page();
float saturation = (float)1.0 - ((mTimeNow - mLastDate) / (float)(48 * 60 * 60 * 1000));
float hue = (float) (new Date()).getHours() / (float) 23.0;
theNewPage.mbaseHue = hue;
theNewPage.mColor = Color.getHSBColor(hue, saturation, (float) .5));
theNewPage.mPath = "0";
theNewPage.mNumHits = 0;
theNewPage.pageCorners = new Corner[maxPoint3Ds];

// set background color of applet
hue = 0.4375;
if (hue > 1.0)
    hue = 1.0;

mBackColor = Color.getHSBColor(hue, (float) .75, (float) 1.0));
this.setBackground(mBackColor);
```

```javascript
const theNewPage = new Page();
let saturation = N.float32(1.0 - (this.mTimeNow - this.mLastDate) / (48 * 60 * 60 * 1000));
let hue = N.float32((new Date()).getHours() / N.
    float32(23.0));
theNewPage.mbaseHue = hue;
theNewPage.mColor = GFX.Color.getHSBColor(hue, saturation, N.float32(0.5));
theNewPage.mPath = "0";
theNewPage.mNumHits = 0;
theNewPage.pageCorners = new Array(this.maxPoint3Ds);
// original comments : set background color of applet
hue = 0.4375;
    hue = N.float32(hue);
if (hue > 1.0) {
    hue = 1.0;
    hue = N.float32(hue);
}
this.mBackColor = GFX.Color.getHSBColor(hue, .75, 1.0));
```

Fig. 7. (left) Screenshot of the original Java source code in John F. Simon Jr.’s Unfolding Object, 2002. This excerpt renders the opening image, before interaction by the user. Courtesy of the Solomon R. Guggenheim Museum.

Fig. 8. (right) Screenshot of the same excerpt as in fig. 7, here resituated into the context of JavaScript. The project programmer, Karl Toby Rosenberg, had to apply only minimal changes to the code excerpt in order to render it functional again. Courtesy of the Solomon R. Guggenheim Museum.

However, even similar languages can feature differences that have to be accounted for during code resituation. In the treatment of Unfolding Object, the project programmer needed to manage the difference in how Java and JavaScript handle floating point numbers, which are numbers with decimal values. It was necessary to simulate the way that floating point numbers are handled in Java so that the results of numeric calculations in JavaScript would remain consistent with those of the original Java. Similar to the treatment approach for Brandon’s <blink> tags, the programmer solved this problem by writing external functions in JavaScript.

Other differences that require compensation when migrating code from one programming language to another include the availability of specific
programming tools and code libraries. In the case of *Unfolding Object*, Java and JavaScript’s different underlying code libraries rendered the graphics in different ways: the artist used the default graphics library in Java for the original artwork, which features a sharp image where each pixel can be seen individually, resulting in staircase-like steps in the pixelation of diagonal lines (fig. 9). In contrast, JavaScript’s default graphics library, Canvas 2D, renders pixelated lines blurry to smooth the staggering effect (fig. 10). In order to recreate *Unfolding Object*’s pixelated graphics with the JavaScript restoration code, the programmer implemented an alternative customized code library using WebGL 1.0 (fig. 11).

![Fig. 9. (left) Screenshot of the original artwork featuring a sharp image where each pixel can be seen individually, resulting in staircase-like steps in the pixelation of diagonal lines. Courtesy of the Solomon R. Guggenheim Museum.](image1)

![Fig. 10. (middle) When using JavaScript’s default graphics library, Canvas 2D, the edges of shapes and lines are blurred. Courtesy of the Solomon R. Guggenheim Museum.](image2)

![Fig. 11. (right) After the replacement of JavaScript’s default library with custom WebGL 1.0, the restoration code programmed by Karl Toby Rosenberg faithfully recreates the pixelated look of the original artwork. Courtesy of the Solomon R. Guggenheim Museum.](image3)

Almost the entire section of *Unfolding Object*’s code that generates the unfolding pages in response to the user’s mouse clicks can be resituated in this way from Java to JavaScript. The concept of code resituation allowed a “copy and paste” approach as a way to migrate much of the source code from Java into JavaScript.
while handling the language differences within the restoration code.

CONCLUSION: CODE RESITUATION, AN APPROXIMATION

The newly developed treatment method of code resituation allowed much of *Unfolding Object*’s original code to be retained, including its original algorithms and other evidence of the artist’s creative expression. By integrating and activating sections of the original, the procedure of code resituation clearly minimizes the intervention, especially if compared with a straightforward code migration that deletes or deactivates original code. However, some of the original code will still be lost or altered when migrating an artwork from one programming language to another. The process of creating a fully functional restoration version can, in some cases, resemble an act of interpretive translation.

The extent to which this interpretation is close and “true” to the original cannot be easily quantified and will always remain an approximation. While the lack of objective assessability may be hard to accept within the established conservation world, subjectivity is identified and discussed as a given parameter in minimal intervention (Muñoz Viñas 2009) and is integral to the conservation of other forms of variable contemporary art in which interpretation is required each time they are installed or performed.

In search of other applications of the concept of approximation that would perhaps justify and support an approach of interpretive translation, the authors looked outside of conservation and found an area of scholarship and practice in cultural heritage that offers many parallels to code migrations for conservation: literary translation. In the field of literary translation, writers translate poetry, prose, and dramatic texts from one natural language to another. In many cases,
translators are also notable writers in their own right, such as Friedrich Schiller’s translation of Shakespeare’s *Macbeth* into German and Samuel Taylor Coleridge’s translation of Schiller’s *Wallenstein* into English, among many others. Interpretation cannot be avoided in literary translation, but that does not devalue the result. Sometimes a “freer” or creative interpretation is necessary to approximate the tone of the language or the expression of an author.

The field of literary translation offers a scholarly and practical framework that could inform the further development and refinement of code resituation practices. The professional field of literary translation includes a code of ethics for translators (Landers 2001); the concept of “linguistic” integrity, which is analogous to the concept of respecting the artist’s intent and preserving the artist’s hand; and an accepted practice on the frequency of repeating translation for a given work. Literary translation regards itself as a means to experience a work in an authentic way. Sometimes, translations have to be updated to continue to serve this purpose. For example, a late 19th century translation of a Tolstoy novel would sound outdated to our ears today—new and updated translations are published regularly of important and well-known works. A further parallel to explore between literary translation and code migration is the role of the living author and the author’s relationship with the translator, which can be analogous to that of a conservator working with a living artist.

Against the background of literary translation, code resituation can be regarded as an effective tool to minimize the need for extensive interpretation by embedding and activating original code. To speak in literary terms, code resituation cites the original, marks the citation, and contextualizes the citation with interpretative language. This interpretation may become temporary, as future conservators and programmers continue to maintain the artwork for the enjoyment of future generations.
ACKNOWLEDGMENTS

The authors would like to thank the artists Shu Lea Cheang and John F. Simon Jr. for their support of this research. Further thanks go out to Jonathan Farbowitz, the Fellow of the Conservation of Computer-based Art at the Solomon R. Guggenheim Museum; the programmer of the Brandon Restoration, Emma Dickson; and the programmer of the Unfolding Object restoration, Karl Toby Rosenberg.

We would further like to thank our colleagues who took time to review this publication: Professors Ernest Davis and Craig Kapp, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University; Prof. Glenn Wharton, Department of Museum Studies, New York University; and Lena Stringari, Deputy Director and Chief Conservator at the Solomon R. Guggenheim Museum.

The CCBA initiative at the Guggenheim is supported by the Carl & Marilynn Thoma Art Foundation; the New York State Council on the Arts with the support of Governor Andrew Cuomo and the New York State Legislature; Christie’s; and Josh Elkes.

REFERENCES


Guggenheim Collection Online. N.d. “Cheang. Shu Lea Cheang, Brandon.”


**CONTACT INFORMATION**

Deena Engel
Clinical Professor
Director, Program in Digital Humanities and Social Science
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, Room 404
New York, NY 10012
212-998-3131
deena.engel@nyu.edu
http://cs.nyu.edu/~deena/

Joanna Phillips
Formerly: Senior Conservator, Time-based Media
Solomon R. Guggenheim Museum, New York
Currently: Director, Conservation Center Düsseldorf
Ehrenhof 3a
D-40479 Düsseldorf, Germany
+49 (0) 211 89 92436
joanna.phillips@duesseldorf.de

Electronic Media Review / Proudly powered by WordPress