EUROGRAPHICS Symposium on Sketch-Based Interfaces and Modeling (2010)
M. Alexa and E. Do (Editors)

Assisted Multitouch Image-Based Reconstruction

Frank Bauer and Marc Stamminger

Computer Graphics Group, University of Erlangen-Nuremberg

Abstract

We present an image-based reconstruction approach for mobile, multitouch enabled devices. A novel scene description based on a multi-agent system is used to allow a real-time reconstruction workflow even on devices with relatively low processing speeds. Using the built in camera along with data from an accelerometer, compass and GPS module allows us to easily add new camera objects to our reconstruction world with an initial estimation for position and orientation. Our multi-agent scene description proved to be flexible enough to perform modelling tasks beyond image-based reconstruction using multitouch gestures only.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Modeling packages

1. Introduction

Latest PDAs or mobile phones include a camera, GPS, an accelerometer and maybe even a compass. Whenever a photo is taken, it is thus possible to also store the camera's world position and orientation. If a particular object is photographed from several positions and directions, we obtain a (roughly) calibrated set of views, which is a perfect basis to construct a textured 3D model from this object, for instance in the style of Facade [DTM96].

The general idea of Facade is simple: the user first provides the basic structure of the object by defining primitives and constraints for their relative position. For instance, the curch's tower in Fig. 1 is composed of a box and a pyramid. The pyramid is always on top of the box, and width and depth of the pyramid and the box are the same. The remaining degrees of freedom (tower's height, roof's height, width and depth) are fixated by marking the position of primitives' corners or edges in images of the object. Global optimization is used to find a solutions for these parameters. Finally, the images are projected back onto the reconstructed geometry to achieve textures.

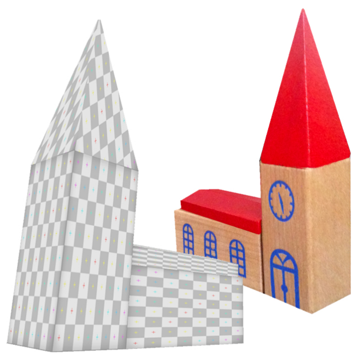


Figure 1: *Reconstructed toy-church created from three images on an iPhone (4 Blocks, 30 Agents)*

In this paper, we describe such a modelling system designed for a mobile device with small display, multitouch screen, and relatively low CPU power. The choice of this setup had a number of interesting consequences for our modelling tool:

- Due to the lack of CPU power and the desire for interactive reconstruction we cannot afford costly high-dimensional global optimization.
- The definition of the primitives and constraints must happen with (multitouch) gestures only.
- For on-site reconstruction, it must be possible to incrementally refine a model and add new images with the possibility to undo improvements for the worse.
- Input images are already coarsely oriented and positioned, so we have a good starting point for optimization.

Users of gesture controlled applications also expect im-

DOI: 10.2312/SBM/SBM10/025-032

mediate feedback to their input. This makes it necessary that our system updates the scene at interactive frame rates as soon as the user triggers a change.

As a result, we developed a reconstruction process which is based on a *multi-agent system* [WW01]. A scene consists of a number of geometric primitives (rectangles, cubes, spheres etc.). The parameters of these primitives (dimensions, world position, rotation, radius etc.) are incrementally set and updated by agents, which enforce constraints such as

- objects A and B both lie on a common plane or
- object A's corner k is visible in image i at position (x, y)

Agents are thus used to enforce user-defined constraints between the primitives, as well as to transform geometry to fit with the images.

In our system, agents always work sequentially. An agent only updates parameters in a way that it does not invalidate previously established constraints. If this is not possible, it tries to minimize the overall error. Agents thus look for previous agents they can interfere with and try to find an optimal common solution using simple local optimization.

Next to agents that monitor constraints, we also have agents that create geometric primitives, or do other modifications on the primitives. The reconstruction process is described by a sequence of agents, that sequentially generate the final model. Since all agents are deterministic, the reconstruction state can always be recomputed by sequentially evoking them from the start.

We found that this way to model the reconstruction process fits very well our needs:

- By the sequential processing of the agents the reconstruction problem is not described as one big high-dimensional problem, thus costly global high dimensional optimization is avoided.
- Agents and thus the entire reconstruction process can be defined using multitouch gestures.
- The reconstruction process can be iteratively refined by adding, reordering, or removing agents.
- An improvement for the worse can always be undone by removing or changing the offending agent recomputing the reconstruction beginning with the changed agent.
- Since user changes only affect a small number of agents, we can reevaluate the scene very fast, providing a true interactive modelling experience.

In fact it turned out that the agent-based description of the reconstruction process is not only beneficial for image-based reconstruction on a mobile device. We also show that it is a user-friendly way to model scenes in any modelling tool.

2. Previous Work

It is a difficult task to generate precise models using multitouch or other finger-based gestures. This drawback is often balanced by additional algorithmic efforts. Murugappan et al. for example did beautify sketches by suggesting prototypes for curves [MSR09]. Masry et al. derive primitives from sketches using a kinematic simulation [MKL07]. In contrast, our system starts with precise primitives (cubes, spheres...). Multitouch gestures are then used to refine that geometry or define constraints.

Constraints are an important part of image-based reconstruction using Facade [DTM96]. We think the general process lends itself for multitouch enabled devices. However the monolithic optimization is not well suited for mobile applications due to CPU restrictions. In general users of multitouch software expect immediate results. This invalidates the two-stage (planning and optimizing) process Facade uses. The tool PhotoTourism [SSS06] finds corresponding points in a multitude of input images. Besides the fact, that many images are needed, the reconstruction itself is quite CPU intense. Moreover the resulting reconstruction is a point cloud that would need further processing.

With procedural modelling, the scene is described by means of simple equations. Mueller et al. derive a procedural, monolithic description for facades in [MZWVG07]. Our application is based on the advantages of a procedural scene description as well, however agents must not be monolithic and have to work independently. This makes them easier to develop and maintain.

As we mentioned earlier, constraints are frequently used (for example: [BB88], [KGC97], [WTJ*03]). Unfortunately most described constraint based systems optimize all equations in one high dimensional global pass and often incrementally build upon previous results. This approach is slow (especially on embedded hardware), the results are often unpredictable and errors produced by subsequent optimization passes can not be easily revised.

3. Multi-Agent Scene Description

All reconstructed scenes are described by a sorted list of agents A and a set of available primitives P. Initially, both are empty. Every user action is translated into a new agent that is automatically added to the end of A. Executing the agents in sequence produces the final scene deterministically. Based on this representation we can optimize the system to work at interactive speeds even on mobile CPUs. Fig. 3 depicts the basic idea.

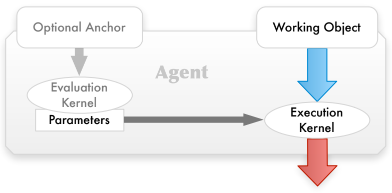


Figure 2: *Structure of an agent*

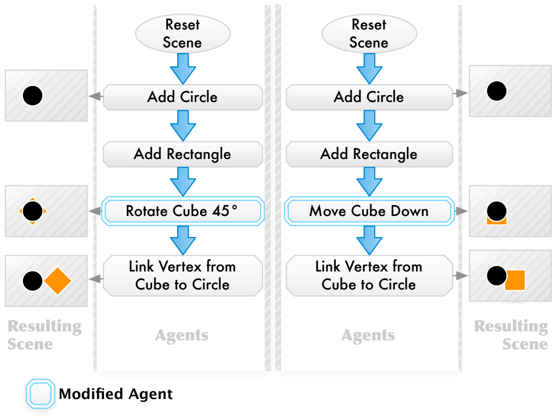


Figure 3: *A multi-agent scene description. The 3rd agent was modified in the right scene producing a different, but consistent result.*

An agent is a lightweight object that performs a specific task without global knowledge. Tasks include

- primitive generation (spheres, cubes, cameras, ...)
- storing user input (dragging, rotation, ...)
- enforcing constraints
- geometry modification (adding vertices, lines, faces, ...)

Each primitive created by an agent is stored in P along with an affine transformation and a set of items like its vertices, lines or faces. Agents are used to enforce constraints, but they are more general and can fulfill any programable task, like adding new faces to an existing primitive.

If an agent is modified, we return to the state immediately before the agent was first executed, alter the parameters of the agent and re-execute all following agents. In Fig. 3 the rotation-agent is replaced with a translation-agent on the right side. The system would handle the change like this:

- reset scene to state after the 2nd agent (*Add Rectangle*)
- modify the 3rd agent
- re-execute agents 3 and 4

The user can modify existing agents at any time while keeping all changes that were applied at a later point. It is easy for users to understand the consequences of changes as the resulting scene is calculated and presented at interactive speeds. In the example from Fig. 3 replacing the rotation with a translation generates a different result that still satisfies the constraint defined by the 4th agent (vertex on circle).

In the following subsections, we present the general concept of our multi-agent scene description. In Sect. 4 we detail on the extension to multitouch image-based reconstruction.

3.1. Agents

Agents are used to execute small programs that alter the scene, one of its primitives or an item of a primitive. Fig- 2 presents the design of an agent i. It stores a parameter vector $\vec{p}_i$ and optionally references an anchor object. The values of $\vec{p}_i$ can be changed by a call of the agent's *Evaluation Kernel* $ev_i(\vec{p})$. The *Evaluation Kernel* may access the referenced anchor object to calculate $\vec{p}_i$. The *Execution Kernel* $ex_i(\vec{p})$ is called, whenever the agent's parameters need to be applied to the scene. In most cases the *Execution Kernel* modifies the working object referenced by the agent.

Fig. 4 demonstrates the execution of a point-on-surface agent. The anchor object is the sphere, the anchor item is its surface (orange circle). The working object is the cube, its item is the selected vertex (purple circle). The agent's parameter vector describes a translation. The *Evaluation Kernel* computes the translation needed to move the cubes to ensure that the vertex is on the surface of the sphere. The *Execution Kernel* changes the transformation matrix of the cube according to the values from $\vec{p}_i$. The result of the agents execution is depicted in the middle of Fig. 4.

The user can manually drag, rotate or scale the cube. The execution part of our multi-agent scene description enforces all constraints. In this case the cube moves along the surface of the sphere. The transparent cube in the rightmost image of Fig. 4 depicts the position before the constraint was satisfied.

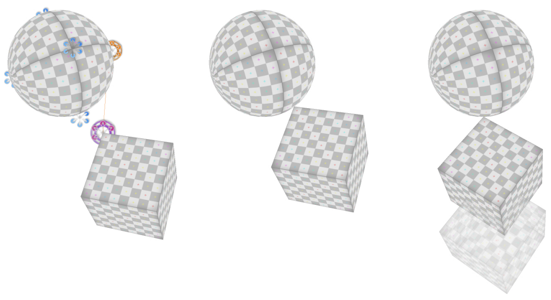


Figure 4: *Creating a constraint (**left**). Scene after the agent was executed (**middle**). Manual modifications to the initial state (**right**).*

3.2. Agent Categories

Agents can be grouped into four basic categories:

Constant Agents: These are the simplest agents. They take a constant value (quaternion, translation-vector, scale, ...) as input and apply it to the working object or one of its items. Another example for this category are agents that create new primitives.

Copy Agents: Copy agents copy values (like the scale) from an anchor object to the agents working object. The *Evaluation Kernel* of agents from this class stores the values from the anchor object in $\vec{p}_i$. The *Execution Kernel* writes the values from $\vec{p}_i$ to the working object.

Constraint Agents: These agents are used to enforce dependencies like the point-on-surface constraint from Fig. 4. In general, an anchor object is needed to compute $\vec{p}_i$ in the *Evaluation Kernel*. The following constraint agents are available in our implementation:

- point-on-point, point-on-line, point-on-surface
- line-on-point, line-on-line, line-on-surface
- surface-on-point, surface-on-line, surface-on-surface

There are other possible agents for this category, for example a line-intersects-line agent. Our system can easily be expanded by new agents.

It is important to note that a point-on-line-agent is different from a line-on-point-agent since agents can only influence their working object.

Modelling Agents: Agents from this category can change the set of available items for an object. As proof of concept we implemented a very simple subdivision agent. It splits a selected line in half, creating new faces if necessary. Fig. 5 shows two subsequent subdivision agents. The resulting new geometry (in the above example two additional vertices, one additional line and one additional face) is used by the following agent, in this case a line moving agent.

Changing the geometry of an object after some agents were already applied to it is unproblematic. The scene is always reset (which undoes the geometry change) and the agents are executed in their original sequence. Agents executed before the geometry modification still work with the items from the original primitive. Later agents may use the altered items. As a consequence, agents that use newly created items cannot be moved to a position before that item was created.

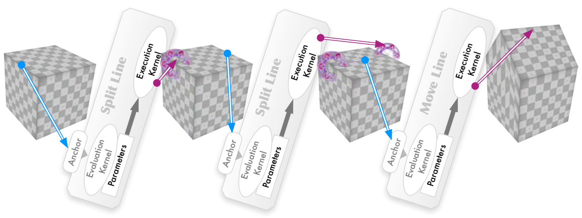


Figure 5: *Scene with line split agents*

3.3. Executing Agents

Every scene is described by a sorted list of multiple agents. Executing them in sequence creates a scene that respects the defined constraints and modifications.

To support fast re-execution of unmodified agents, we separated the calculation of the parameter values $\vec{p}_i$ (*Evaluation Phase*) from the execution of the agent. The two phases of the re-execution process are depicted in Fig. 7.

During the *Evaluation Phase* the *Evaluation Kernel* $ev_i(\vec{p}_i) : \vec{p}_i \rightarrow \vec{m}_i$ is called for every agent. Agents with constant parameters (most agents without an anchor object/item) can update $\vec{p}_i$ within the *Evaluation Kernel*. If $\vec{p}_i$ depends on non constant values (like the position of an anchor object), we need to compute it in the kernel. After the *Evaluation Kernel* has finished, the *Execution Kernel* is called. The later call makes sure that the results of the agents evaluation are reflected in the scene.

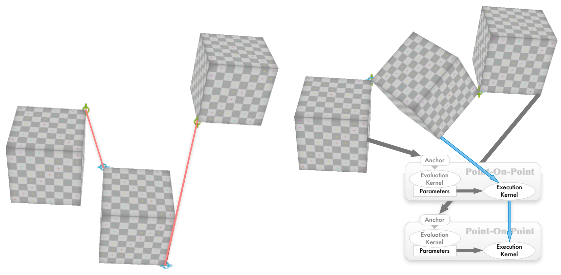


Figure 6: *Scene before agents (depicted by red lines) are executed (**left**). After execution of both agents (**right**).*

A point-on-point agent for example depends on the current state of the anchor to calculate the parameter values. The example in Fig. 6 shows two vertex constraints (red lines) for the middle cube. Calculating a translation vector can satisfy the first one. Moving the cube again to enforce the second agent can satisfy the later one, but violates the results of the first. The second agent needs to calculate a rotation and an additional translation to satisfy both constraint. This results in a non-linear system of equations [Sca85].

To solve the non-linear system, the *Evaluation Kernel* returns a measurement vector $\vec{m}_i$. If $||\vec{m}_i|| = 0$ the agent is satisfied. We use the measurements in a Levenberg-Marquardt solver [Mor77] that updates the agent's parameter vector $\vec{p}_i$ in each iteration. Rotations - traditionally a problem in non-linear optimization - are represented as quaternions and reparameterized for the minimization process as detailed in [SN01].

If $\vec{p}_i$ is computed directly in the *Evaluation Kernel*, the dimension of $\vec{m}_i$ is 0. That way, it does not contribute a measurement (as it would always be 0 anyway) for the non-linear solver.

When two agents (like in Fig. 6) apply to the same working object the later may invalidate the result of the first. Agents need to be aware of previously executed agents for the same object, and make sure that they do not interfere. To integrate this behaviour into our system, we introduce the concept of a *Solver Group*.

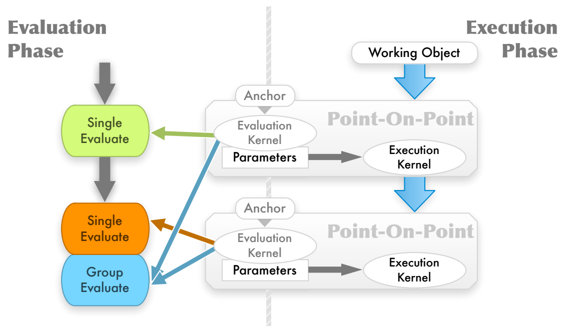


Figure 7: *Schematic execution of agents from Fig. 6 (right)*

The *Solver Group* S_i for a seed agent i contains all previous agents that use the seed agent's anchor as working item. If we resolve cycles, we additionally use the anchors of all added agents to determine participation in S_i.

Computing $\vec{p}_i$ for an agent i during the evaluation phase is a three step process:

1. Run $ev_i(\vec{p}_i)$. If the dimension of $\vec{m}_i$ is not 0 start a solver run with measurements $ev_i(\vec{p}_i)$.
2. Start a solver run with measurements $\vec{m} = \bigcup_{k\in S_i} ev_k(\vec{p})$ where $\vec{p} = \bigcup_{k\in S_i} \vec{p}_k$.
   This potentially changes the parameters of all participating agents, but ensures that no previous agent is violated. The state of the scene immediately before i was executed is used as initial input when solving the non-linear system of equations.
3. Run $ex_k(\vec{p}_k)$ for $k \in S_i$ (modifies the state of the scene).

Calling the *Executing Kernels* of all agents in sequence after the *Evaluation Phase* produces the correct (or in cases where not all constraints can be fulfilled, the best possible) result.

Our incremental approach exploits the weakness of non-linear solvers. When the user adds a new constraint, we assume that the next best solution is the closest local minimum of the current equation system. If this is not the case, the user can manually drag the primitives to a better starting point by inserting a new agent directly before the constraint. The user can transform the object, until the following optimizer falls into the correct minimum. This behaviour helps preventing erratic jumps in the resulting scene when only minor modifications are applied, something we often did observer when using a global optimization approach.

3.4. Scene Changes

With our multi-agent system, only a limited number of agents need to be reevaluated if part of the scene changes. When a new agent is added, evaluating and executing the newly created agent (as well as all agents in its *Solver Group*) is sufficient.

Changes to the input of an existing agent are a bit more complex to process. We need to reexecute the changed agent and all later agents that use the modified working object as either anchor or working object. This is accomplished by a minor adjustment to the *Evaluation Phase*.

The working object of the changed agent is marked as dirty. We run the *Evaluation Phase* beginning with the changed agent, leaving the results of all previous agents untouched. Before the *Evaluation Kernel* of an agent is called, we check the dirty flag of its anchor and working object. If one is set, the agent is reevaluated and its working object marked dirty.

During our reconstruction tests, the average of agents executed was 5.3 (Scenes contain an average of 35 Agents).

This approach is possible, because we can reset our scene to any previous state. A global approach (where all constraints are solved in one big equation system) would always reevaluate the complete scene. In addition, most global approaches use the result of the latest solved system as the input of the next solver pass, making it impossible to undo wrong decisions at a later stage.

3.5. Parameter Selection

Most constraint agents alter different parameters $\vec{p_w}$ of the referenced working object w to enforce their constraint. The example from Fig. 4 could also be solved by changing the cube's scale instead of translation. Optimizing for an unnecessary high dimensioned $\vec{p_w}$ often confuses the Levenberg-Marquardt solver and produces an erratic changing $\vec{p_w}$. The result is an inconsistent scene. Our agents use a simple test to decide which parameters they should use to satisfy their constraints.

When an agent is executed for the first time, we test all possible parameter combinations and choose the one combination that results in the least error. We cache this result to save performance time for subsequent executions. However, if the error generated by the cached parameter assignment is greater than a given threshold, we try to find a better set.

Object parameters $\vec{p_w}$ (like the objects' translation, rotation or scale) used by an agent are locked, and cannot be used by later agents. If no more parameters are available for a new agent, it will choose from all available parameters.

4. Image-Based Reconstruction

Our multi-agent system is flexible enough to be used as a basis for an image-based reconstruction tool. To this end, our system offers a special camera object. The user can select feature points in the camera image. These are translated into a modelling agent that adds the ray from the focal point

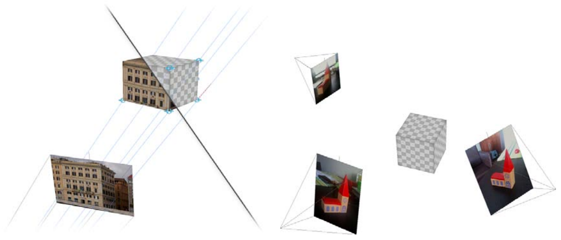


Figure 8: *Basic image-based reconstruction (**left**). Automatic camera orientation (**right**)*

of the camera through the marked feature point to the camera primitive. The reconstruction itself is then as simple as adding constraints between an object an those camera rays.

Fig. 8 (left) shows a very basic reconstruction using 5 feature points. The resulting lines were used as anchors in 5 point-on-line constraints for the cube object.

4.1. Retrieving Cameras

The estimation of the initial camera position and rotation is often problematic. Modern cameras store GPS information and some even their orientation. This data can be used to estimate the initial position of the camera in the virtual scene. On an iPhone for example, we access the GPS information as well as the current compass and accelerometer reading. When our software takes a picture with the built-in camera, we create a new virtual camera object and set the position and orientation according to the gathered values.

The first camera is always placed in the origin of the virtual world. If subsequent images are in close proximity to the first camera (determined by the recorded GPS-Information), we assume that the new camera looked at the centre of the scene, and position it accordingly based on its recorded orientation (Fig. 8 right).

If the distance to the first camera is greater than a given threshold, we use the GPS-Information to determine the relative position of the camera in the virtual world.

When the application runs on an iPad or a Mac, we can connect it to a remote iPhone. An image taken on the connected device automatically shows up with the estimated position and orientation.

The information gathered by the GPS-Sensor and the accelerometer is precise enough for an initial estimate. The compass, however, is very fragile, and responds to very small magnetic changes in the environment.

4.2. Calculating Global Orientation

An accelerometer returns a gravity vector $\vec{G}$, relative to the devices' local coordinate system. By its nature an accelerometer cannot record rotations around the gravitation axis. As a result it is impossible to determine which direction the device was looking at if it is held upright. This is why we also need a compass. The compass returns a vector $\vec{M}$ pointing to the magnetic north pole. Using both we can determine an absolute orientation for the device.

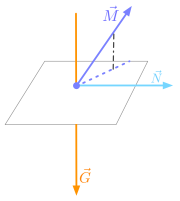
On an iPhone we calculate the rotation around $\vec{G}$ by projecting $\vec{M}$ into the plane perpendicular to $\vec{G}$ (Fig. 9, left). The angle between calibrated north $\vec{N}$ and the projection of $\vec{M}$ is the rotation around $\vec{G}$. The vector $\vec{G}$ should point in the inverse direction of the normal from the devices screen. We use this information (along with the magnetic rotation) to calculate the absolute rotation of the device.

4.3. Feature Selection

When tapping the camera with two fingers (instead of one), the camera image is displayed full screen. The view can be set to three different modes: Creation, Modification and Pick.

In *Pick-Mode* a feature-point can be picked to create a new constraint. *Modification-Mode* allows the points to be moved around the image. In *Creation-Mode* new feature-points can be added.

When defining features on a multitouch device, the user needs some assistance. One common approach is to magnify the screen region where the users fingers are pointing. The placement of this magnification is critical for its usability. It must be presented in a region of the screen that is not obscured by the users hand.



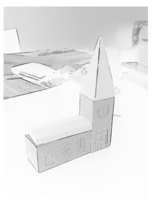


Figure 9: *Rotation around the gravity vector (**left**). Result of edge enhancement (**right**)*

To assist the user further, we implemented an edge detection. We use multiple filter and recognition passes on the input image and combine them to the final edge enhance image (Fig. 9 right). The following images are blended using the specified percentage as weight:

- Grayscale image (20%)
- Canny filtered image (30%)
- Lines from a Hough transformation (50 %)

The edge image is used as an alternative view in the magnification. If the user wishes, a corner detection algorithm locates the edge closest to the input and moves the feature point to that location. This allows subpixel exact positioning of feature points even on multitouch devices.

5. User Interface

With multitouch enabled gestures, users can navigate a 3D-scene with ease. A basic overview is given in [Sel08]. In our implementation a single finger move rotates the scene around its centre. Moving two fingers translates the scene, and using a pinch-gesture zooms in and out. Picking is intuitive as well using a single tap. Tapping with two fingers opens the view of the last used camera object. For our reconstruction task, we need to provide additional gestures to allow object transformations and constraint creation.

Most tests were performed on an iPhone 3G-S. The physical size of the devices screen makes it impossible to display all necessary controls at once. To compensate this restriction, our App uses context aware glyphs and overlays.

When an object is picked, it is augmented by several glyphs that provide controls for tasks like moving, rotating and item picking. Fig. 10 (left) shows these glyphs. When an item is selected it can either be modified (Fig. 10, right) or it can be the working item of a new constraint. In that case we also need to pick the anchor item in the same manner. Fig. 4 (left) demonstrates the creation of a new constraint. The cube is the working object. The vertex marked with the purple circle is the working item. The sphere (augmented by the picking-glyphs) is the anchor and the line marked with the orange circle is the anchor item. This creates a constraint that ensures that the selected vertex stays on the spheres surface.

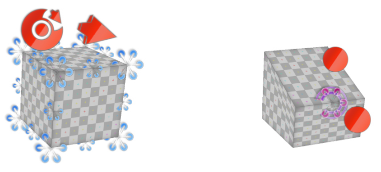


Figure 10: *Cube augmented by pickable glyphs (**left**). Moving the marked (purple circle) line (**right**).*

Together these concepts provide a very natural reconstruction experience. The concept of inter object constraints allows for precise models and is very useful in the context of multitouch reconstruction.

Once agents are created, they are stored in a sorted list. We allow the user to modify that list in our user interface. This way we can delete unneeded or faulty agents or change the order the agents are executed in. Reordering agents might have an effect on the resulting scene, as most agents use the scene state created by previous agents to calculate their parameters in the *Evaluation kernel*.

6. Results

We built three different scenes. The toy church from Fig. 1 was reconstructed on an iPhone 3G-S. We used multiple cameras to enhance the textures for different views. All images were taken with the iPhone and automatically positioned. The derived orientations in the scene were very close to the ones reconstructed (less than 1% off). The position information was not used, as the camera locations were to close. The images were uncalibrated.

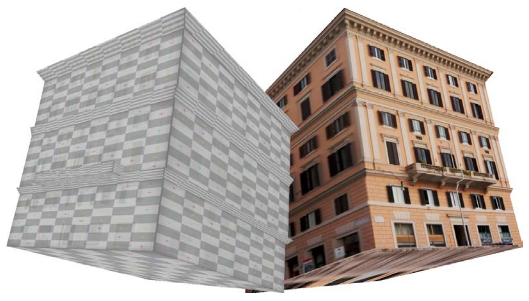


Figure 11: *Reconstruction of a house in Rome*

Fig. 11 shows a simple building. It is created from 7 primitives and one camera. The model in Fig. 12 shows a wooden Viking Church. It is composed from 13 cube primitives some modified using line-split agents to resemble a roof. A single camera was used.

As we mentioned in the introduction, our system also provides a very user-friendly way to model simple scenes with nothing but multitouch gestures. Fig. 13 shows a model of a figure created with our system on an iPhone. Moving the spheres attached to the arms, also rotates and stretches that arm. This behaviour is implicit in our system as executing all agents in sequence simply enforce the constraints that were used to model the figure.

The reconstruction tests on an iPhone show that for most changes the speed of the iPhone CPU is sufficient. Most user interactions are additions of new primitives or agents. In that case, the number of agents we need to execute is small compared to the number of total agents. In rare cases, when a

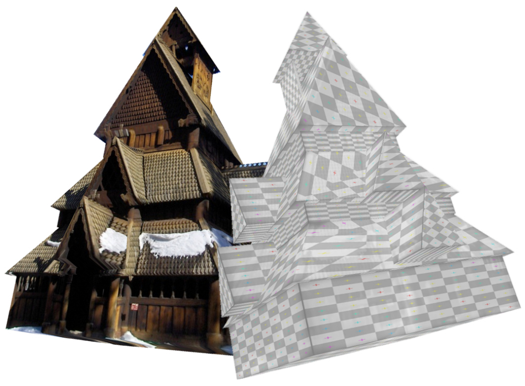


Figure 12: *Reconstruction of a Viking Church*

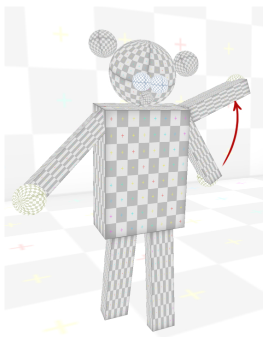


Figure 13: *Model with basic inverse kinematics*

modified agent alters an object that is used as anchor in many other agents, the speed is not sufficient. Since every agent takes a different amount of time (depending on the initial parameter values and the type of the agent), it is difficult to specify the maximum number of agents we are able to execute at interactive frame-rates. In our practical test the average limit was around 15-20 agents.

For most tasks, our system scales much better than the traditional global optimization. When a user enhances the scene, we only need to re-evaluate a very small subset of agents with a low dimensional parameter vector. In the case of global optimization we would need to reevaluate the complete set of constraints for all available parameters resulting in a noticeable slower reconstruction for large scenes.

7. Conclusions

The results of our test on the iPhone show, that a global, high dimensional optimization would be impractical with limited CPU power. Through our design we work around this limitation, by only evaluation a few lightweight agents most of the time. Only in rare cases do we have to execute all or a sufficient high number of agents that the user will experience lag.

Reconstruction using only multitouch gestures, without any help from a keyboard or high precision pointing devices feels quite natural. The software is easy to understand and simple to use. The features of modern devices - like accelerometer, GPS or compass - make it very easy to estimate the initial position of a camera, which is of great importance for a working reconstruction.

Describing the scene by a multi-agent system proofed to be even more flexible than we initially thought. It is quite simple to enhance the system with new agents, as they each perform a local and very small program (in our experience max. 10 lines of code). Together, they provide a system powerful enough to perform many kinds of modelling tasks. This includes surface reconstruction of point clouds, which we did not discuss in this paper. Especially the *-on-point agents are very useful in that line of work.

The size of the scene for reconstruction on an iPhone is limited - foremost by the very small screen. Devices like the iPad or Microsoft's Surface are better suited. Both platforms could leverage the benefit of our system, offering bigger screens and faster CPUs.

References

[BB88] BARZEL R., BARR A. H.: A modeling system based on dynamic constraints. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), ACM, pp. 179–188. 2

[DTM96] DEBEVEC P. E., TAYLOR C. J., MALIK J.: Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM, pp. 11–20. 1, 2

[KGC97] KWAITER G., GAILDRAT V., CAUBET R.: Interactive constraint system for solid modeling objects. In *SMA '97: Proceedings of the fourth ACM symposium on Solid modeling and applications* (New York, NY, USA, 1997), ACM, pp. 265–270. 2

[MKL07] MASRY M., KANG D., LIPSON H.: A freehand sketching interface for progressive construction of 3d objects. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), ACM, p. 30. 2

[Mor77] MORÉ J. J.: The Levenberg-Marquardt algorithm: Implementation and theory. In *Numerical Analysis* (Berlin, 1977), Watson G. A., (Ed.), Springer, pp. 105–116. 4

[MSR09] MURUGAPPAN S., SELLAMANI S., RAMANI K.: Towards beautification of freehand sketches using suggestions. In *SBIM '09: Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling* (New York, NY, USA, 2009), ACM, pp. 69–76. 2

[MZWVG07] MÜLLER P., ZENG G., WONKA P., VAN GOOL L.: Image-based procedural modeling of facades. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers* (New York, NY, USA, 2007), ACM, p. 85. 2

[Sca85] SCALES L. E.: *Introduction to non-linear optimization*. Springer-Verlag New York, Inc., New York, NY, USA, 1985. 4

[Sel08] SELKER T.: Touching the future. *Commun. ACM 51*, 12 (2008), 14–16. 7

[SN01] SCHMIDT J., NIEMANN H.: Using quaternions for parametrizing 3-d rotations in unconstrained nonlinear optimization. In *VMV '01: Proceedings of the Vision Modeling and Visualization Conference 2001* (2001), Aka GmbH, pp. 399–406. 4

[SSS06] SNAVELY N., SEITZ S. M., SZELISKI R.: Photo tourism: exploring photo collections in 3d. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM, pp. 835–846. 2

[WTJ*03] WILCZKOWIAK M., TROMBETTONI G., JERMANN C., STURM P., BOYER E.: Scene modeling based on constraint system decomposition techniques. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision* (Washington, DC, USA, 2003), IEEE Computer Society, p. 1004. 2

[WW01] WOOLRIDGE M., WOOLDRIDGE M. J.: *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001. 2