A Component-based approach towards Mobile Distributed and Collaborative PTAM

Tim Verbelen† Pieter Simoens† Filip De Turck‡ Bart Dhoedt§

Ghent University - IBBT, Department of Information Technology
†Ghent University College, Department INWE
§

ABSTRACT
Having numerous sensors on-board, smartphones have rapidly become a very attractive platform for augmented reality applications. Although the computational resources of mobile devices grow, they still cannot match commonly available desktop hardware, which results in downscaled versions of well known computer vision techniques that sacrifice accuracy for speed. Given the heterogeneity in mobile devices, applications also have to be tailored to the specific capabilities of each device. To cope with these issues, we propose a component-based approach towards mobile augmented reality applications, where components can be configured and distributed at runtime. This allows for device specific configurations of the same software, and for a performance increase by offloading CPU intensive tasks to a server in the network. In this paper, we present a component-based implementation of the Parallel Tracking And Mapping (PTAM) algorithm, together with a middleware framework to transparently configure and distribute components at runtime. This leads to a mobile, distributed version of the original PTAM algorithm, as well as several extensions and a collaborative scenario.

1 INTRODUCTION
Component-based development is not a new approach towards augmented reality (AR). Already in the early 2000s the need grew to create AR frameworks that offer components providing common functionality such as marker tracking, scene graph management and rendering [2], with the main objective to enable developers to quickly prototype and create new AR applications on top of such a framework. By distributing some of these framework components (i.e., a distributed scene graph), this evolved to collaborative AR [10], where multiple users can see and interact with the same virtual objects.

When mobile devices such as personal digital assistants (PDAs) became available, AR research shifted towards handheld AR, and mobile AR frameworks were proposed [14]. To increase the tracking performance, the tracking process was executed on a server connected to the mobile device via WiFi [16].

Nowadays, smartphones are becoming more and more the preferred platform for mobile AR, as they are equipped with cameras, GPS sensor, accelerometers. Although their computational resources are not up to par with common desktop or server infrastructure, they do have enough processing power to allow real-time detection and tracking for augmented reality [15]. However, to cope with the limited processing power on the mobile device, these algorithms are often heavily modified versions of known state-of-the-art algorithms, sacrificing accuracy and quality to gain speed [6] [15]. Also, some algorithms such as large scale 3D reconstruction still use too many resources and have to be outsourced to a remote server [1].

Instead of deciding at design time which parts of the AR application should run on the mobile device and which should run on a remote server, we propose to adopt a component-based approach that decouples software components from their execution location. These components are configured and distributed at runtime, which allows the application to adapt for different heterogeneous mobile devices. Component-based development also makes it easy to extend applications by adding components, enables efficient reuse of existing components, and transparently allows collaborative applications by sharing components between different users.

In this paper, we present a component-based implementation of the Parallel Tracking And Mapping (PTAM) algorithm [5], a widely used tracking algorithm in a priori unknown environments. Our middleware framework implemented in Java allows to run components on Android, the most popular mobile OS [3], as well as on commodity hardware such as desktops or servers. We illustrate the three main advantages of our component-based middleware. First we show how the middleware framework can transparently distribute components, allowing to offload and speed up CPU intensive components. Second, we extend the PTAM algorithm by adding and reusing components: an object recognition component allows to annotate and track known objects in the map similar to [4] and a better relocalization algorithm is presented, both using Lowe’s SIFT features [8]. Third, we present a collaborative PTAM algorithm, where multiple users track the same map, and keyframes from all users are used to update and refine the map, which is easily achieved by sharing components.

In the next section, we first show how to split up the PTAM algorithm in different components, and how these components communicate. Section 3 discusses the design and implementation of the component management middleware. In Section 4 we show why different component configurations are needed for different mobile devices, and how to increase performance by outsourcing components from the mobile device to a remote server. Section 5 presents two extensions to the PTAM algorithm, and how we can easily integrate them in the application by adding components. Section 6 presents a collaborative PTAM algorithm, where multiple users can concurrently track and extend the same map. Finally, Section 7 concludes this paper and future work is discussed.

2 COMPONENT-BASED PTAM
The PTAM method as proposed by Klein et al. [5] is shown on Figure 3. The tracker thread processes every camera frame, assuming a 3D map of feature points is given. Each image is sampled in a 4-level pyramid, and at each scale corner features are found using the FAST-10 detector [12]. A prior estimate of the camera pose is calculated using the previous pose and a constant velocity filter. Using this prior estimate, all potentially visible 3D map points are projected into the image. Then matches are sought between the projected map points and the found corner features, from which a new
camera pose is estimated using 10 iterations of reweighted least squares optimization. The matching is done in two stages: first a coarse tracking stage, where only features in the highest pyramid levels are used, and a second fine tracking stage where features from all levels are reprojected. When not enough matches are found, tracking is considered lost, and a relocalization procedure is initiated. Finally a decision is taken whether a new keyframe should be added to the map on the spatial coverage of the scene, and the camera position is used to render a 3D overlay on the frame.

The mapper thread continuously updates and refines the map. When a new keyframe is added, all map points are projected into the keyframe using the pose estimated from the tracking process. New map points are instantiated from unmatched features in regions of the keyframe where no matches with existing map points are found. By performing an epipolar search in neighboring keyframes, 3D positions can be triangulated. After a keyframe is added to the map, a local bundle adjustment is performed on the added keyframe and its four nearest neighbors. Next a global bundle adjustment is run on all the keyframes and map points. The map is initialized by the user, who chooses the first two keyframes by carefully moving the camera, allowing features to be tracked between the two frames. From these two keyframes and pairs of matched features a relative camera pose can be calculated and an initial set of 3D map points is triangulated.

When tracking is lost, relocalization is initiated by creating a small blurry version of the image. This small blurry image is matched via the sum of squared differences with all the small blurry images of the keyframes in the map. The camera position is then estimated from the 3D information of the best matching keyframe, together with a 2D rotation and translation by aligning the blurry image with its best match in a least square sense.

From this algorithm outline, we distilled 5 components as shown in Figure 2:

**VideoSource** The VideoSource fetches video frames from the camera hardware. These frames are analyzed by the Tracker, and rendered with an augmented reality overlay by the Renderer.

**Renderer** Each camera frame is rendered on screen together with an overlay of 3D objects. These 3D objects are aligned according to the camera pose given by the Tracker.

**Tracker** The Tracker searches map points in video frames and calculates the new camera pose. When a new keyframe should be added to the map, this keyframe is sent to the Mapper.

**Relocalizer** When not enough map points are found in the video frame, the Tracker calls the Relocalizer, that estimates the camera position using the small blurry image approach.

**Mapper** The Mapper receives keyframes from the Tracker, that are used to initialize and extend the map. The mapper also performs the bundle adjustment to optimize the map points. Other components such as the Tracker and Relocalizer register themselves as MapListeners, that receive notifications when the map is updated.

These components are implemented based on the source code of [5]. The resulting application is shown in Figure 3. On the right a grayscale video frame is shown with the tracked feature points, from which the camera position is estimated. The left shows the resulting overlay with a 3D object, and a white border around a recognized book, which is an extension of the algorithm presented in Section 5.

![Figure 3: The augmented reality application tracks feature points in the video frames (right) to enable the overlay of 3D objects (left).](image)

### 3 Component management middleware

In order to develop and run component based applications, a component model is needed for defining components. Also, a middleware framework is needed that can start and stop components, manage their lifecycle, bind components together and set up the communication between them, etc. In order to enable distributed and collaborative PTAM, our component middleware has three important requirements: portability, transparency and performance. In order to distribute components, portability is needed to allow the component code to be executed on different devices with possibly different processor architectures. Transparency means that components should be able to communicate, regardless on which machines they are deployed. Because the application has to process camera frames in real-time, performance is also of key importance.

An overview of our middleware framework is shown in Figure 4. As base for the component-based middleware, we use OSGi [13], a service oriented module management system in Java, allowing to dynamically load, unload and configure software components – called bundles – at runtime. OSGi bundles can expose service interfaces by registering implementations of these interfaces with the OSGi service registry. Other bundles can query the service registry for available service interfaces, or can listen for a requested service to come online. The portability of Java enables the execution of
presented a series of adaptations to cope with the high CPU re-
slow on the iPhone.
performed on the coarser resolution images, which leads to a loss
fewer map points are used and the number of keyframes in the ma-
of measurement precision. To lower the cost of bundle adjust-
config for each device, or can even be adapted at runtime by the middleware, so that on lower end devices the application can still process frames in a timely manner at the cost of some tracking accuracy.
To show the impact of these parameters on the tracking performance, we measured the time needed to track a frame for different parameters configurations on two Android devices. The first de-
second device is the LG Optimus 2x, which is powered by the Tegra 2 processor that has two cores clocked at 1 GHz. We compare two input resolutions: 800x480 pixels, which is the screen resolution of both devices, and 400x240 pixels.

the same code on different platforms and architectures, facilitating remote execution and code migration.
Because Java code is still slower than native C/C++ code, especially on the Android platform [7], native code is preferably used for performance critical applications such as augmented reality. Therefore, we allow components to embed native libraries, that can be called from Java code via the Java Native Interface (JNI). Allowing native libraries poses a trade-off on portability versus performance. To still end up with a component that can run on multiple heterogeneous devices, the developer should embed different versions of the native library, compiled for all the supported architectures and operating systems. The middleware will then select and load the right native library at runtime.

To transparently distribute OSGi bundles, our middleware generates a proxies of the service interfaces provided by different components. When other bundles request a service, the OSGi framework will return a reference to our proxy, which forwards the method call to the proxied bundle, or to a remote instance of the bundle by a remote procedure call (RPC). To look up service interfaces of remote instances R-OSGi [11] is used, which besides distributed service lookup, also offers the RPC protocol to call remote services.

4 Mobile, Distributed PTAM
To enable the PTAM algorithm on the iPhone 3G, Klein et al. [6] presented a series of adaptations to cope with the high CPU requirements. For faster tracking, the search for feature points is only performed on the coarser resolution images, which leads to a loss of measurement precision. To lower the cost of bundle adjustment, fewer map points are used and the number of keyframes in the map is kept at a minimum, and also the stereo map initialization was too slow on the iPhone.

Instead of limiting the tracking to the coarse images, the Tracker component adopts the same algorithm as the PC version of PTAM, but a few parameters are made configurable to tailor the Tracker to the available CPU resources for each device. In the fine tracking stage, the number of points reprojected and sought in the image can be limited to a lower threshold (1000 in the desktop version). Also, the FAST feature detection stage can be omitted, in which case the features are matched using zero-normalized SSD score of a 8x8 template in a 5-pixel-radius. Another important factor on the tracking performance is the resolution of the input images, which can be configured in the VideoSource component. These parameters can

Figure 5: The time needed to track a frame as a function of the number of feature points reprojected, for two different devices, two different image sizes and with or without FAST corner detection stage.

Figure 5 shows the effect of omitting the FAST corner detection stage and the number of reprojected features on the time needed to track a frame. The tracking time per frame linearly increases with the number of feature points searched, and a higher image resolution requires more processing. This clearly shows that a different configuration is needed for each device and the optimal configuration depends on the desired amount of frames processed per second. For example, if one desires a tracking framerate of 15 frames per second, on the Optimus 2x an input resolution of 800x480 and a search for 400 points suffices, while on the Desire the input res-

To mitigate the high computational cost of the mapping process, and in particular the global bundle adjustment, while still allowing map sizes comparable to the PC version, we can offload the Mapper component to a PC or laptop connected to the mobile device via a wireless connection. Because the mapping is a background process, the delay of sending a camera frame to the server has little impact on the user experience, while this speeds up the global bundle adjustment, as shown on Figure 6, where the local execution on the Optimus 2x is compared to offloading a laptop with a dual core 2.26GHz processor connected via WiFi, resulting in a speedup of a factor 10.
The component-based approach not only allows the application components to be distributed among devices in the network, component-based development also enables efficient reuse and extendability of the application. To illustrate this, we extended the PTAM algorithm with two features. First, we added an object recognition and localization algorithm to annotate recognized objects in the scene using the triangulation of SIFT features, as proposed by Castle et al. [4]. Second, we implemented a new Relocalizer component, that instead of matching small blurry images, also uses SIFT feature matches with the keyframes in the map to relocalize the camera. For example, when the camera is rotated compared to the camera position in the nearest keyframe as shown in Figure 7, the small blurry image (SBI) method will not be able to correctly relocalize the camera position.

Because the SIFT features are scale, rotation and illumination invariant, the SIFT relocalization algorithm is still able to find matches between the current image and the keyframe as shown on Figure 8. The extended application architecture is shown in Figure 9. There are now two relocalizers: the previously discussed Small Blurry Image (SBI) Relocalizer and the new SIFT Relocalizer. The following components are new and colored in gray:

**SIFT Feature Detector** The SIFT Feature Detector takes an image as input and returns an array of feature point locations and their SIFT feature descriptors [8].

**Feature Matcher** The Feature Matcher matches a number of input feature descriptors to a database of known feature descriptors. Features are matched using randomized kd-trees as proposed by [9].

**Object Recognizer** The Object Recognizer listens to map updates and searches the keyframes for known objects. For each keyframe the SIFT feature descriptors are calculated and matched against a database of known features, using the two previous components. If the number of features matching with an object exceeds a threshold, that object is marked as seen in the keyframe. Once an object is detected in two or more keyframes, its location can be triangulated as described in [4], and notified to the Renderer that shows a bounding box or an annotation.

**SIFT Relocalizer** The SIFT feature points can also be used for better relocalization when tracking is lost. The SIFT Relocalizer calculates SIFT feature descriptors for each keyframe added to the map, and those feature descriptors are added to the database of the Feature Matcher. When tracking is lost, the SIFT Relocalizer calculates feature descriptors of the current camera frame, and matches the features against the ones of the keyframes of the map. A camera position can then be calculated when sufficient matches are found.

Although SIFT features descriptors are computationally complex to calculate, we can still use these components in a mobile application, again by offloading them to resources in the network. For example on the Optimus 2x calculating SIFT features in a 800x480 frame takes about 20 seconds, while on the laptop this is about 2 seconds.

**5 EXTENDING PTAM**

The component-based approach not only allows the application components to be distributed among devices in the network, component-based development also enables efficient reuse and extendability of the application. To illustrate this, we extended the PTAM algorithm with two features. First, we added an object recognition and localization algorithm to annotate recognized objects in the scene using the triangulation of SIFT features, as proposed by Castle et al. [4]. Second, we implemented a new Relocalizer component, that instead of matching small blurry images, also uses SIFT feature matches with the keyframes in the map to relocalize the camera. For example, when the camera is rotated compared to the camera position in the nearest keyframe as shown in Figure 7, the small blurry image (SBI) method will not be able to correctly relocalize the camera position.

**Figure 7:** When tracking is lost, using the SBI method no matches are found (right) when the camera position is rotated compared to the closest known keyframe (left).

**Figure 8:** Although the camera is rotated, still enough SIFT matches are found to estimate the position from the new frame (right) compared to the keyframe (left).

**Figure 9:** By adding new components (in gray) the PTAM algorithm can be easily extended. The SIFT Feature Detector and Feature Matching components extract and match SIFT feature descriptors. These components can be used for an enhanced SIFT based Relocalizer, or an Object Recognizer that annotates detected object in the AR overlay.
deployed on a PC connected to the wireless network. All users have a device running a VideoSource, Renderer, Tracker and Relocalizer component, and connect to the PC running the Mapper. The middleware will transparently bind the services together, resulting in a collaborative user experience.

![Diagram](image_url)

**Figure 10:** By sharing components (in gray) between multiple users collaborative AR applications can be composed. Sharing the Mapper component allows multiple users to track and extend the same map. By splitting up the Renderer in a Model-View-Controller pattern and sharing the Model component, multiple users can see and interact with the same augmented 3D scene.

Next to the map, we also want to have a shared state of the virtual 3D objects the users can interact with. To accomplish this, we split up the Renderer component according to the Model-View-Controller pattern. The state of the virtual world is captured in the Model component. All actions of the user, e.g. a button press, are handled by the Controller, that will forward the action to the Model. When the Model is changed, the Renderer is updated to render the updated model. By splitting up the Model in a separate component, the state of the virtual 3D objects can also be shared among multiple users for a true collaborative application. The resulting architecture is shown in Figure 10, with the shared components colored in gray. Figure 11 shows how two mobile devices collaboratively track and update the map.

![Collaborative PTAM](image_url)

**Figure 11:** Collaborative PTAM on two mobile devices: the map is updated and refined on the laptop with keyframes from the two devices. Both devices track the map and render the shared scene graph.

### 7 Conclusion and Future Work

In this paper we presented a component-based middleware platform that configures and distributes software components transparently at runtime. We showed the effectiveness of this approach for augmented reality applications, using a component-based implementation of the PTAM algorithm. Comparing different configurations of the Tracking component for different devices shows the need for device specific configurations. By transparently offloading components to a server connected via WiFi, performance can be increased and the algorithm can be extended with other resource intensive components. By sharing components between multiple users, a collaborative PTAM algorithm is achieved.

As future work we want to develop autonomic algorithms for the middleware platform, which allows the middleware to detect performance problems at runtime, and to reconfigure or distribute components in order to achieve an acceptable user experience. This would allow developers to write their applications as a number of configurable components, and the middleware will tailor the application for each device at runtime and choose the best configuration and/or distribution.

### Acknowledgements

Tim Verbelen is funded by Ph.D grant of the Fund for Scientific Research, Flanders (FWO-V).

### References


