Flikken: Programming a Mobile Game with Context-Aware Tuples

Elisa Gonzalez Boix, and Christophe Scholliers
{egonzale | cfscholl}@vub.ac.be

Software Languages Lab
Vrije Universiteit Brussel, Belgium

Abstract. This document describes the implementation of Flikken, a mobile game implemented using context-aware tuples in which players equipped with mobile devices interact in a physical environment augmented with virtual objects. Coordination and interaction between players is fully specified by means of context-aware tuples and reactions to some of these tuples e.g. to show on the GUI the virtual objects spread around the city. We describe the game functionality and the context-tuples used in its implementation. Before concluding, we provide code snippets for the relevant parts of the game implementation.

This document assumes the reader understands the Context-Aware Tuples model and knows the operations it provides.

1 Description

Flikken[1] is a mobile game implemented using context-aware tuples inspired by an industrial-strength game called The Target[2]). Flikken is a so-called virtually augmented game in which players equipped with mobile devices interact in a physical environment (i.e. a city) augmented with virtual objects. Players are organized in two teams which determine their role in the game. The policemen work together to shoot down a dangerous gangster on the loose before he achieves his goal of earning 1 million euro by committing crimes. In order to commit a crime and get the reward, the gangster needs to collect burgling equipment around the city (e.g knives, explosives, guns, etc.).

Figure 1 shows the gangster's and a policeman's mobile device at the time the gangster has just burgled the local casino. As shown in the screenshots, the gangster knows the location of places with big amounts of money (e.g banks, casinos, etc.). When a gangster commits a crime, policemen are informed of the crime location and the amount of money stolen. Policemen, on the other hand, can see the position of all nearby policemen. Further they can send messages to

[1] Flikken is available for download as a context-aware tuples program with the AmbientTalk language at htttp://soft.vub.ac.be/amop.

[2] http://www.lamosca.be/en/the-target

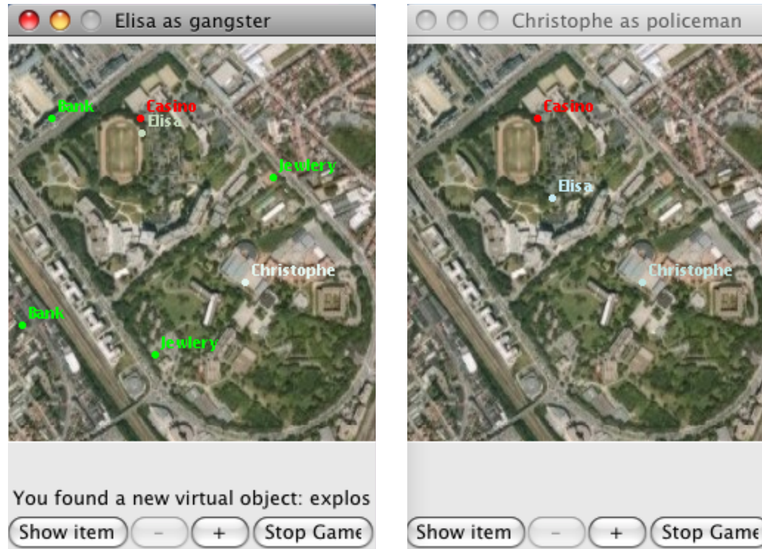

Fig. 1.

each other in order to coordinate their movements. Both the gangster and policemen are periodically informed of each other positions. Policemen and gangster can also shoot at each other. At the beginning of the game the gangster gets four additional virtual objects for his defense: a mine (which kills anybody who steps on it), a gas bomb (which kills anybody in a determined radius for a period of time), a radio jammer (which disrupts the connectivity of the nearby policemen, preventing them to know his location) and a bulletproof vest (which protects him for a time interval against one single shot).

Flikken is an ideal case study for context-aware tuples as it epitomizes a mobile ad hoc networking application that has to react to context changes on the environment such as changes on player's location, appearance and disappearance of other players, or the discovery of virtual objects while moving about. Moreover, how to react to these changes highly depends on the receivers of the contextual information, e.g. virtual objects representing burgling items should only be perceived by the gangster when he is nearby the item's location while they should not be perceived at all by policemen. In the remainder of this section, we illustrate how the Flikken game uses context-aware tuples. Coordination and interaction between players is fully specified by means of context-aware tuples and reactions to some of these tuples e.g. to show on the GUI the virtual objects spread around the city. Thanks to the language symbiosis capabilities of AmbientTalk, our application can call upon Java for GUI construction. Hence, the game GUI is implemented in Java. Before diving into the discussion of the distributed coordination aspects of the game we briefly describe the setup.

2 Implementation

In contrast to The Target, Flikken do not rely on a centralized server to play the game. When a player starts the game, a CAT system is created in his mobile device. This CAT system is used for all distributed aspects of the game, e.g., to communicate periodically his positions to the opposite team. At the start of the game, policemen and gangster are waiting in their headquarters. A headquarter (HQ) also has a tuple space which injects set of tuples to the players: the gangster's HQ injects four tuples representing the above mentioned weapons that the gangster has for his defense, while the policemen's HQ injects 3 tuples representing the charge of their guns. Each policeman gets three bullets at the start of the game but they can reload their guns by returning to their HQ. The chase starts once the gangster leaves his headquarter. Throughout the city various context providers (i.e. CAT systems) are placed to inform the gangster about virtual objects (additional weapons or burgling equipment) or crime locations by injecting the necessary tuples. Once the chase starts, policemen and the gangster communicate player to player by means of the CAT network.

Table 1 shows an overview of the tuples used in the game and its context rule. The tuples are divided in five categories depending on the entity which injects them in the environment, i.e. all players, only gangster, only policemen or city context provider (which includes the ones injected by headquarters). A tuple is denoted by the term τ. As usual, the first element of a tuple indicates its type. We use capitals for constant values.

The CAT system on the player's device carries a vital private tuple $\tau(TeamInfo, ?name, ?team)$ indicating to which team he belongs. Every player transmits its location to the CAT network by means of the tuple $\tau(PlayerInfo, ?name, ?team, ?location)$. These tuples are often used in other tuple's context rules to identify the current whereabouts of a player and his team. The context rule is defined by a set of templates and constraints on those templates which needs to be satisfied. For example, the context rule for the `VirtualObject` tuple indicates that this tuple goes in context when the receiver is a gangster whose location (extracted from the `PlayerInfo` tuple) should be in communication range with the virtual object. As a concrete example, consider the following `VirtualObject` tuple for a granade.

```
cat.inject: tuple(VirtualObject, grenade)
inContext: [tuple(TeamInfo, ?u, GANGSTER),
            tuple(PlayerInfo, ?u, GANGSTER, ?loc),
            inRange(grenadeLocation, ?loc)]
```

The tuple $(VirtualObject, grenade)$ should be only perceived if the receiver is a gangster whose location (given by `?loc` in the `PlayerInfo` tuple) is in physical proximity with that virtual object. The `inRange` function builds the constraint that checks if the gangster location is in euclidian distance with the location of the grenade (stored in the `grenadeLocation` variable). Upon removal of a `VirtualObject` tuple, a private tuple $(hasVirtualObject, ?object)$ is inserted in his CAT system. `hasVirtualObject` tuples are used in the context rule of `CommitCrime` tuples which notify the gangster of a crime that can

Tuple Content	Tuple Context Rule	Tuple Description
	All Players	
τ(TeamInfo, uid, gip)	n/a	Private tuple denoting the player's team.
τ(PlayerInfo, uid, gip, location)	[τ(TeamInfo, ?uid, ?team), ?team $\neq$ gip]	Injected every 6 minutes to notify the position of a player to opposite team members. Location is a 2-tuple indicating the (GPS) coordinates of the player.
τ(InHeadquarters, location)	[τ(PlayerInfo, ?u, ?team, ?loc), inRange(location, ?loc)]	Notifies HQ that the player moved in its communication. Used to detect the start of the chase (when this tuple goes out of context for the HQ of the gangster) and the arrival of policemen to their HQ (to reload guns).
	Only The Gangster	
τ(CrimeCommitted, name,location, reward)	[τ(TeamInfo, ?uid, POLICEMAN)]	Notifies policemen that the gangster committed a crime.
	Only Policemen	
τ(PlayerInfo, uid, gip, location)	[τ(TeamInfo, ?u, gip)]	Notifies the position of a policemen to his colleagues.
	City Context Providers	
τ(VirtualObject, id, location)	[τ(TeamInfo, ?u, GANGSTER), τ(PlayerInfo,?u, GANGSTER,?loc), inRange(location, ?loc)]	Notifies the gangster of the nearby presence of a virtual object. `inRange` is a helper function to check that two locations are in euclidian distance.
τ(CrimeTarget, name, location)	[τ(TeamInfo, ?u, GANGSTER)]	Notifies the gangster of the position of crime targets.
τ(CommitCrime, name,location, reward, vobj)	[τ(TeamInfo, ?u, GANGSTER), τ(PlayerInfo,?u, GANGSTER,?loc), inRange(location,?loc), hasVirtualObjects(vobj)]	Notifies the gangster of the possibility of committing a crime. `hasVirtualObjects` takes an array of virtual object ids and checks that the gangster has the required `VirtualObject` tuples.
τ(VirtualObject, BULLET)	n/a	Represents the bullets of the gun of gangster's and policemen's gun. The gangster only gets three at the start of the game, while policemen can get three more when they return to their HQ.

Table 1. Overview of the Context Aware Tuples used in Flikken

be committed. As crimes can only be committed when the gangster has certain burgling items, the context rule of the `CommitCrime` tuple requires that certain $VirtualObject$ tuples are present in his CAT system. For example, in order for the gangster to perceive the `CommitCrime` tuple for the `grandCasino`, a $(hasVirtualObject, grenade)$ tuple is needed as shown below.

```
cat.inject: tuple(CommitCrime, grandCasino, location, reward)
inContext: [tuple(TeamInfo, ?u, GANGSTER),
            tuple(PlayerInfo, ?u, GANGSTER, ?loc),
            inRange(location, ?loc),
            tuple(hasVirtualObject, grenade)];
```

Each player also registers several reactions to (1) update his GUI (e.g. to show the `hasVirtualObject` tuples collected), and (2) inject new tuples in response to the perceived ones, e.g. when a gangster commits a crime, he injects a tuple $(CrimeCommited, ?name, ?location, ?reward)$ to notify policemen. The code below shows the reaction on `PlayerInfo` tuples installed by the application.

```
def makePlayer(username, team) {
  ...
  cat.whenever: tuple(PlayerInfo, ?uid, ?tid, ?location) read: {
     GUI.displayPlayerPosition(tid, uid, location);
  } outOfContext: {
     //grey out a player if there exists no update of his coordinates.
     def tuple := cat.rdp(tuple(PlayerInfo, uid, tid, ?loc));
     if: (nil == tuple) then: { GUI.showOffline(uid) };
  };
};
```

Whenever a `PlayerInfo` tuple is read, the player updates his GUI with the new location of that player. As `PlayerInfo` tuples are injected with a timeout, they are automatically removed from the tuple space after its timeout elapses triggering the `outOfContext:` handler. In the example, this handle greys out the GUI representation of a player if no other `PlayerInfo` tuple for that player is in the CAT system. If the `rdp` operation does not return a tuple, the player is considered to be offline as he did not transmit his coordinates for a while.

2.1 Evaluation

Flikken demonstrates how context-aware tuples aid the development of context-aware applications running on mobile ad hoc networks and address the tuple perception issues shown in section **??** by introducing two key abstractions: (i) the context rule of a tuple which determines the conditions that a receiving CAT system should have in order to perceive the tuple, and (ii) the rule engine which takes care of inferring tuple perception before applications can access a tuple. A tuple space model with such abstractions has the following benefits.

1. Decomposing a tuple into content and context rule leads to separation of concerns (i.e. increased modularity).
2. Since context rules can be developed separately, it enables programers to reuse the rules to build different kinds of tuples (i.e. increased reuse). For example, in Flikken, we used a `inRangeOfGangster(?loc)` function to build the rule for the different `VirtualObject` tuples which was also reused to build the three first conditions of `CommitCrime` tuples.
3. Programmers do not need to add computational code to infer tuple perception as the rule engine takes care of it in an efficient way, making the code easier to understand and maintain.

Without context-aware tuples, tuple perception needs to be computed at the application after a tuple is read. Determining that a tuple is appropriate for the context situation of the receiver is significantly complex due to two reasons. First, programmers need to register numerous event handlers to manually infer tuple perception. More concretely, programers need to install event handlers to observe the presence or absence of *every* tuple that is currently used in the context rules. Those event handlers may be triggered independently requiring complex code to compose them together. Such an explicitly event-driven architecture is difficult to program, understand and maintain [1]. Secondly, the content of the tuple has to be polluted with meta data to be able to infer tuple perception. For example,

the `VirtualObject` tuple should also contain the team information, which is clearly not part of its functionality. In conclusion, programmers need to write complex computation code which actually deals with coordination in order to compensate the lack of expressiveness of the coordination model.

References

1. P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002.