Akademia Górniczo-Hutnicza
Im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej
Katedra Informatyki Stosowanej

Jan Jędrychowski
Łukasz Spas

Graficzne interfejsy aplikacji opartych o biblioteki Qt i KDE

Praca Dyplomowa

Promotor: dr inż. Igor Wojnicki

Kraków 2012

Oświadczam, świadoma(y) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałam(em) osobiście i samodzielnie, i że nie korzystałam(em) ze zródeł innych niż wymienione w pracy.

Jan Jędrychowski
Łukasz Spas

Spis treści

Spis rysunków

Spis tablic

Rozdział 1

Wstęp

W dzisiejszych czasach coraz bardziej powszechne staje się wykorzystanie przeglądarek do zadań, do których wcześniej używane były duże aplikacje klienckie. Powstają rozwiązania, które starają się oddzielić logikę obliczeniową od warstwy prezentacji, przenosząc jednocześnie tę pierwszą na stronę serwera. Rozwój technologii HTML5 rozszerzającej standard o elementy canvas, websocket, webworkers i inne umożliwia tworzenie aplikacji o możliwościach takich samych jakie niegdyś były dostępne tylko w programach desktopowych. Co więcej gwarantuje międzyplatformowość nie tylko w rozumieniu softwareowym – jedna aplikacja dostępna jest zarówno na komputerach osobistych, tabletach, telefonach i innych urządzeniach wyposażonych w nowoczesną przeglądarkę. Przy użyciu bardzo związanej z HTML5 technologii CSS3 możliwie jest tworzenie jednej aplikacji, która będzie użytkowalna niezależnie od wielkości ekranu urządzenia.

W niektórych rozwiązaniach zastąpienie starych aplikacji desktopowych nowymi aplikacjami webowymi (przeglądarkowymi) jest jednak niemożliwe, czasochłonne lub zbyt kosztowne.

Podczas badań rynku pod kątem aktualnie dostępnych technologii dostrzeżono braki w rozwiązaniach umożliwiających zdalną interakcję z pojedynczymi aplikacjami. Większość z rozwiązań dostępnych na rynku wymusza udostępnienie całego pulpitu oraz wymaga od użytkownika końcowego (klienta) posiadania odpowiedniego, nierzadko płatnego oprogramowania (np. TeamViewer, VNC, Citrix, X11 i inne). Celem projektu jest stworzenie alternatywy wymagającej od strony klienta jedynie przeglądarki obsługującej HTML5 bez konieczności instalacji jakichkolwiek pluginów (np. Java, Flash).

Głównym wzorcem dla tej pracy jest projekt GTK+ Broadway powstały w 2011 roku oferujący dostęp przez przeglądarkę internetową do aplikacji działających pod kontrolą biblioteki GTK na zdalnym serwerze. Do tej pory nie istniało rozwiązanie oferujące podobną funkcjonalność dla biblioteki Qt i stworzony na potrzeby tej pracy projekt jest pierwszą taką implementacją. Kluczowym czynnikiem wyróżniającym tę pracę na tle innych jest innowacyjny sposób przesyłu danych do wizualizacji okien i ich elementów, który nie opieraja się na transmisji bitmap.

// TODO: Jak? // TODO: Po co?

Rozdział 2

Podstawy teoretyczne

W rodziale tym przedstawione zostaną najważniejsze informacje dotyczące technologii wykorzystanych w projekcie.

2.1 Wybrane rozwiązania HTML5

HTML5 (ang. HyperText Markup Language) jest najnowszą wersją popularnego języka znaczników HTML. Pojęcie to nie jest do końca jasne i oczywiste, ponieważ ta edycja języka niesie ze sobą nie tylko zmiany w znacznikach, ale bardzo mocno rozszerza możliwości stron WWW. Co więcej łączy się bezpośrednio z innymi technologiami takimi jak Javscript oraz CSS3 i nie jest w stanie bez nich istnieć. W związku z tym sama definicja HTML jako jedynie język znaczników jest niepełna. We wcześniejszych etapach samo konsorcjum W3 miało problemy z jasną definicją HTML5 i na krótki czas składowymi tej technologii był język CSS3 oraz SVG. Standard nie jest jeszcze ukończony i zgodnie z zapowiedziami W3C zostanie ukończony około roku 2014. HTML5 jest rozwijany w ścisłej współpracy z twórcami najpopularniejszych przeglądarek. Została powołana specjalna grupa WHATWG (Web Hypertext Application Technology Working Group), która skupia producentów takich jak Mozilla Foundation, Google, Opera Software oraz Apple Inc. Przeglądarki internetowe takie jak Mozilla Firefox, Google Chrome oraz Opera już teraz implementują większość z planowanych nowości przedstawionych w aktualnym szkicu w wersjach produkcyjnych. Z powodu dojrzałości obecnej formy standard oraz wielkiej popularności już na obecną chwilę można założyć, że jego podstawowe założenia oraz komponenty pozostaną w obecnej formie bez rewolucyjnych zmian.

W proponowanym rozwiązaniu, po stronie klienta zaadoptowano dwa nowe komponenty HTML5: canvas (ang. płótno) oraz WebSocket.

2.1.1 Element canvas

Nowy element drzewa DOM canvas pozwala na renderowanie dynamicznych bitmap na stronie przy pomocy skryptów języka Javascript. Aktualnie wszystkie przeglądarki producentów z WHATWG implementują obecny standard w pełni poprawnie. Wprowadzenie tego komponentu pozwala na tworzenie dowolnych animacji oraz grafik, których użycie wcześniej wymagało użycia zewnętrznych pluginów (np. Flash lub Java). W projekcie element ten używany jest do rysowania pojedynczych widgetów.

2.1.2 Technologia WebSocket

WebSocket jest technologią oferującą ustandaryzowaną pełną dwustronną komunikację między klientem (przeglądarką internetową) a serwerem. Podobną funkcjonalność można było wcześniej zasymulować przy pomocy modelu Comet korzystającego z długotrwałych połączeń HTTP, na które leniwie były wysyłane dane. Poprzednie rozwiązanie z powodu braku ustandaryzowania oraz wykorzystywania obejścia było trudne w utrzymaniu oraz nie oferowało synchronicznej komunikacji dwustronnej. W projekcie technologia wykorzystywana jest do komunikacji z serwerem. Łączność ta jest dwustronna.

2.2 Opis biblioteki Qt

Qt jest międzyplatformowym (ang. cross-platform) frameworkiem aplikacyjnym, najczęściej używany w tworzeniu oprogramowania z graficznym interfejsem użytkownika. Dodatkowo biblioteka zawiera moduły wspomagające między innymi:

- międzyplatformowe *API*[1] dostępu do systemu plików,
- dostęp do relacyjnych baz danych,
- manipulacja XML,
- międzyplatformowe zarządzanie wątkami,
- międzyplatformowe wsparcie dla sieci.

Aplikacje *Qt* tworzone są w języku C++ rozszerzonym o dodatkowe słowa kluczowe i makra, których obsługą zajmuje się program moc (Meta-Object Compiler). Największym uzupełnieniem wniesionym do języka przez framework jest system sygnałów i slotów.

Qt wspiera największe platformy takie jak:

[1] ang. Application Programming Interface

- Windows
- Windows CE
- Symbian
- OS X
- X11 (Linux, FreeBSD, Solaris, AIX i inne)
- Maemo, MeeGo

Framework w wersji 5, która jest akutalnie w fazie beta, ma być dostępny również na wszystkie popularne platformy mobilne takie jak Android, iOS i Windows 8.

W tym podrozdziale zostaną przedstawione mechanizmy biblioteki *Qt* wykorzystane przy tworzeniu projektu, o którym stanowi niniejsza praca.

2.2.1 System zdarzeń

W *Qt* zdarzenia są obiektami dziedziczącymi po klasie *QEvent*, reprezentującymi zajście pewnego zjawiska wewnątrz aplikacji lub będącymi wynikiem oddziaływania z zewnątrz, o którym aplikacja powinna wiedzieć. Zdarzenia mogą być przetworzone przez wszystkie obiekty dziedziczące po klasie *QObject*, która dostarcza podstawowej struktury i logiki niezbędnej do ich obsługi.

Kiedy system operacyjny generuje sygnał o zajściu pewnego zdarzenia, *Qt* dokonuje jego konwersji na odpowiedni i platformowo niezależny format. Każde zdarzenie jest następnie przekazywane do *kolejki zdarzeń* odpowiedniego wątku. Kolejka przechowuje i w odpowiednim momencie rozdysponowywuje zdarzenia do odpowiadających im obiektów odbiorców poprzez wywołanie metody *QObject::event()* wewnątrz której następuje decyzja dotycząca dalszego przetwarzania, zależna od rodzaju zdarzenia.

Niektóre zdarzenia, takie jak na przykład *QMouseEvent* czy *QKeyEvent* pochodzą bezpośrednio od systemu operacyjnego. Inne, jak na przykład *QTimerEvent* czy *QPaintEvent* pochodzą z innych źródeł, nierzadko z wnętrza samej aplikacji (np. do komunikacji między wątkami). Warto w tym miejscu zaznaczyć, że rysowanie w *Qt* nie jest operacją wywoływaną przez system operacyjny lecz przez samą aplikację oraz rysowanie z wnętrza obsługi zdarzenia *QPaintEvent* jest jedynym sposobem na renderowanie graficznego interfejsu aplikacji. Pociąga to za sobą pewne problemy opisane w dalszej części pracy.

2.2.2 System widgetów

Widget'em w bibliotece *Qt* nazywamy obiekt reprezentujący elementy graficznego interfejsu użytkownika takie jak przyciski, listy rozwijane, menu, okna i inne. Klasa

QWidget[2] jest typem bazowym dla wszystkich widgetów i udostępnia niezbędne metody dotyczące renderowania oraz obsługi zdarzeń dzięki czemu w łatwy sposób można uzyskać dostęp do całego interfejsu aplikacji.

Interfejs użytkownika w aplikacjach opartych o framework *Qt* tworzy strukturę hierarchiczną powiązanych ze sobą obiektów klasy QWidget. Wykorzystując ten fakt w łatwy sposób można odtworzyć tą strukturę w innych technologiach, np. tworząc identyczną strukturę w języku HTML. Fakt ten został wykorzystany w niniejszej pracy.

2.2.3 System rysowania

Rysowanie w bibliotece *Qt* standardowo zostało zaimplementowane dla rysowania na ekranie oraz urządzeniach drukujących wykorzystując natywne *API* systemu operacyjnego, dla którego dana wersja *Qt* została skompilowana. Moduł ten jest niejako opakowaniem dla wywołań systemowych, ujednolicając jego logikę i umożliwiając pełną przenośność aplikacji. Na rysunku 2.1 przedstawiony został kaskadowy model systemu rysowania w *Qt*. Jest to *model trójwarstwowy* i każda z klas ma swoje określone zadanie w całym procesie renderowania. Główną zaletą takiego podejścia jest ujednolicenie przepływu procesu rysowania dla różnych urządzeń wyjściowych oraz umożliwienie łatwego sposobu dla dodawania nowych funkcjonalności.

Klasa *QPainter* udostępnia jednolity interfejs umożliwjający wykonywanie operacji rysowania różnych obiektów takich jak linie, okręgi, prostokąty, obrazy oraz umożliwia zastosowanie różnego rodzaju przekształceń, styli czy transformacji macierzowych.

Klasa *QPaintDevice* stanowi abstrakcję dla dwuwymiarowej przestrzeni na której obiekty klasy *QPainter* mogą wykonywać operacje rysowania. Udostępnia ona różnego rodzaju informacje dotyczące specyfiki urządzenia wyjściowego, które mogą być wykorzystane np. do optymalizacji procesu rysowania.

Klasa *QPaintEngine* udostępnia interfejs, za pomocą którego obiekty klasy *QPainter* będą mogły wykonywać operacje rysowania na różnego rodzaju urządzeniach wyjściowych. Klasa *QPaintEngine* jest używana wewnątrz klas *QPainter* oraz *QPaintDevice* i jest ukryta przed aplikacjiami dopóki programista nie zechce stworzyć obsługi dla nowego rodzaju urządzenia wyjściowego. W niniejszej pracy taki właśnie scenariusz został wykorzystany.

// TODO: Obrazek w formie wektorowej

Rysunek 2.1: Schemat budowy systemu renderowania w bibliotece *Qt*

[2]http://doc.qt.digia.com/qt/qwidget.html

2.3 GTK Broadway

Rozdział 3

Określenie problemu i proponowane rozwiązanie

// TODO: Wymagania (ogólne) // TODO: Przypadki użycia // TODO: Zachowanie systemu / software'u // TODO: Struktura systemu / software'u

Przedmiotem pracy jest stworzenie prototypowego serwera hostującego desktopowe aplikacjie oparte o biblioteki *Qt* oraz KDE. Na żądanie klienta serwer uruchamia wybraną aplikację oraz wstrzykuje kod odpowiedzialny za komunikację klienta z procesem aplikacji i przesyłanie klientowi danych dotyczących wyglądu graficznego interfejsu aplikacji.

Postawione zadanie w głównej mierze polega na rozwiązaniu trzech podstawowych problemów:

1. komunikacja między klientem a serwerem,
2. komunikacja między klientem a aplikacją,
3. uzyskanie informacji o wyglądzie elementów graficznego interfejsu aplikacji,
4. symulacja interakcji użytkownika z interfejsem aplikacji.

3.1 Komunikacja między klientem a serwerem

Do realizacji tego zadania stworzony został prosty serwer działający w oparciu o protokół *HTTP*. Jako zasób domyślny udostępnia on listę dostępnych aplikacji, które klient może uruchomić. Lista ta jest w pełni konfigurowalna po stronie serwera. Inicjalizacja połączenia polega na wysłaniu przez klienta nazwy wybranej aplikacji. Serwer po pomyślnej weryfikacji przydziela klientowi unikatowy identyfikator sesji, uruchamia proces aplikacji i wysyła klientowi skrypt w języku JavaScript zajmujący się przetwarzaniem po stronie klienta.

3.2 Komunikacja między klientem a aplikacją

// TODO: Opis do cz. teoretycznej (pogrubione)

Do rozwiązania tego problemu konieczne jest utworzenie ciągłego kanału komunikacyjnego między klientem a procesem aplikacji, za pomocą którego będzie możliwe przesyłanie informacji o wyglądzie interfejsu aplikacji oraz informowanie aplikacji o zdarzeniach generowanych przez użytkownika po stronie przeglądarki. Jako, że za cel przyjęte zostało założenie o nieingerowaniu bezpośrednio w kod skompilowanych już aplikacji, postawiono na technikę umożliwiającą załadowanie kodu biblioteki dynamicznej do przestrzeni pamięciowej procesu aplikacji tuż przed jego uruchomieniem. Kod ten ma za zadanie utrzymanie połączenia oraz transmisję danych między klientem a aplikacją.

3.3 Uzyskanie informacji o wyglądzie elementów graficznego interfejsu aplikacji

Każdy element graficznego interfejsu aplikacji (*QWidget*) jest renderowany w momencie odebrania zdarzenia *QPaintEvent* z kolejki zdarzeń głównego wątku aplikacji. Dzięki temu istnieje łatwy sposób na uzyskanie informacji o tym kiedy oraz ktory element należy przerenderować aby uaktualnić jego wygląd po stronie klienta. Problemem w dalszyb ciągu pozostaje jednak sposób na uzyskanie informacji o samym wyglądzie.

Proponowane rozwiązanie polega na zaimplementowaniu abstrakcyjnego urządzenia wyjściowego reprezentującego przeglądarkę WWW po stronie klienta (patrz podrozdział 2.2.3). Odpowiednio implementując klasy *QPaintEngine* oraz *QPaintDevice* możliwe staje się uzyskanie szczegółowych informacji dotyczących wygądu widgetów co z kolei umozliwia stworzenie innowacyjnego formatu przesyłanych danych. Zamiast przesyłać bitmapy z wyrenderowanym elementem można wysłać informację o kolorach, punktach, liniach i innych podstawowych elementach, które zostaną narysowane na urządzeniu docelowym jakim po stronie klienta jest przeglądarka WWW z obsługą elementów *canvas*.

// TODO Komentarz Co to znaczy innowacyjnego?

3.4 Symulacja interakcji użytkownika z interfejsem aplikacji

Interakcja użytkownika z aplikacją sprowadza się do obsługi następujących zdarzeń:

1. ruch myszy nad elementem,
2. wciśnięcie, zwolnienie oraz dwuklik przycisku myszy,

3. zmiana położenia kółka myszy,
4. wciśnięcie oraz zwolnienie klawiszy na klawiaturze,
5. zmiana rozmiaru okna aplikacji poprzez przeciąganie jego krawędzi,
6. zamknięcie, minimalizacja lub maksymalizacja okna aplikacji.

Większość z wyżej wymienionych elementów jest obsługiwana jako zdarzenia w języku JavaScript większości dzisiejszych przeglądarek. Proponowane podejście na rozwiązanie tego zagadnienia polega na stworzeniu formatu danych bazując na notacji JSON (JavaScript Object Notation). Dane w tym formacie przesyłane do serwera są następnie poddawane walidacji i konwersji na obiekty zdarzeń biblioteki *Qt*. Zdarzenia takie są następnie przesyłane do kolejki zdarzeń w głównym wątku aplikacji.

Odbiorcą zdarzenia jest widget, który wygenerował dane zdarzenie po stronie przeglądarki bazując na hierarchicznej budowie interfejsu użytkownika. Wyjątkami są tutaj zdarzenia klawiatury, które nie mają bezpośredniego odbiorcy w momencie ich zaistnienia. Aplikacja sama decyduje o tym, który element powinien odebrać zdarzenie. Domyślnie jest to widget, który atualnie posiada tzw. focus, a to z kolei zależy od poprzednich zdarzeń oraz logiki samego programu. W celu symulacji podobnego zachowania decyzja o odbiorcy zdarzeń klawiatury podejmowana jest po stronie serwera bazując na aktualnym stanie aplikacji.

// TODO: Przepływy danych / d. kolaboracji (nie wiem co to drugie ma znaczyć :P)

3.5 Zabezpieczenie serwera

Ponieważ jednym z celów projektu było umożliwienie uruchamiania pełnoprawnych aplikacji zainstalowanych na systemie operacyjnym serwera, kluczową staje się możliwość blokowania nieautoryzowanego dostępu do wrażliwych lub potencjalnie niebezpiecznych aplikacji.

W związku z powyższym, stworzono mechanizm list *ACL* (ang. Access Control Lists), który pozwala administratorowi systemu na zdefiniowane, które aplikacje mogą być uruchamiane przez klientów, a w przypadku których zostanie wyświetlony komunikatu o braku dostępu.

Listy kontroli dostępu przechowywane są w pliku konfiguracyjnym serwera w postaci danych w formacie *XML*[1]. Nazwy aplikacji w postaci komend linii poleceń mogą więc być definiowane ręcznie w dowolnym edytorze tekstowym lub za pomocą pliku wykonywalnego serwera poprzez poniższe argumentów wywołania programu:

- *accept-all* spowoduje zniesienie wszystkich wcześniej wprowadzonych obostrzeń i możliwe będzie uruchomienie wszystkich aplikacji zainstalowanych na serwerze,

[1](ang. Extensible Markup Language

- *reject-all* spowoduje zablokowanie wszystkich zapytań serwera. Komenda ta powinna stanowić pierwszy krok w etapie budowy list dostępu,
- *accept nazwa-aplikacji* [2] spowoduje, że aplikacja o podanej nazwie będzie mogła być uruchamiana przez serwer,
- *reject nazwa-aplikacji*, komenda blokujaca możliwość uruchamiania aplikacji o podanej nazwie.

Serwer, ze względów bezpieczeństwa, od razu po zainstalowaniu domyslnie blokuje wszystkie zapytania klientów i oczekuje się od administratora serwera skonfigurowania list *ACL* według uznania. Zmiana ustawień serwera wymaga jego ponownego uruchomienia.

[2]Nazwa aplikacji oznacza pełną komendę wiersza poleceń (wraz z możliwymi argumentami), która spowoduje uruchomienie aplikacji. Może to być również ścieżka bezwzględna do pliku wykonywalnego aplikacji.

Rozdział 4

Implementacja

4.1 Renderowanie

TODO

- OPIS KLASY WebRenderer
- POJEDYŃCZY CYKL RENDEROWANIA.

4.2 Protokół wymiany danych

Aby umożliwić renderowanie elementów po stronie klienta należało utworzyć wspólny format danych bazując na wejściu ze strony biblioteki Qt oraz potrzebnych danych wyjściowych dla obiektu Canvas w języku HTML5.

Każda komenda rysowania po stronie klienta składa się z podstawowych informacji dotyczących rysowanego obiektu, takich jak: identyfikator, pozycja czy rozmiar, oraz listy prostych elementów z których danych obiekt jest złożony (linie, prostokąty, etc.). Poniżej przedstawiono format pojedyńczej komendy rysowania wraz z wszystkimi możliwymi elementami budującymi widgety w aplikacjach opartych o bibliotekę *Qt*.

```
{
  "command":"draw",
  "widget":{
    "id": 12431,           // Identyfikator
    ["z": 0,]              // Pozycja na stosie obiektow potomnych
    "name":"QLineEdit",    // Nazwa obiektu
    "flags": 0x1029,       // Flagi obiektu (definiuja jego typ
                           // i wlasciwosci)
    "x": 100,              // Pozycja X
    "y": 120,              // Pozycja Y
    "w": 200,              // Szerokosc
    "h": 150,              // Wysokosc
    "r":{                  // Renderowany obszar elementu
```

```
      "x": 0,              // Pozycja obszaru X
      "y": 0,              // Pozycja obszaru Y
      "w": 200,            // Szerokosc obszaru
      "h": 150,            // Wysokosc obszaru
    }
  },
  "render":[]              // Lista elementow do narysowania
}
```

Listing 4.1: Komenda renderowania elementu interfejsu

Poniżej w kolejności alfabetycznej przedstawiono elementy, z których może być zbudowany każdy obiekt klasy *QWidget*. Każdy opis zawiera deklarację metody podklasy *QPaintEngine* wykorzystywanej po stronie serwera, format przesyłanych danych oraz opis metody interpretacji tych danych po stronie klienta.

4.2.1 Elipsy

```
virtual void QPaintEngine::drawEllipse( const QRectF & rect );
virtual void QPaintEngine::drawEllipse( const QRect & rect );

{
  "t":"ellipse",
  "x":0.0,        // Pozycja srodka X
  "y":0.0,        // Pozycja srodka Y
  "w":10.0,       // Srednica pozioma
  "h":10.0        // Srednica pionowa
}
```

Pełna implementacja za pomocą krzywych Beziera. Kod implementacji:

```
// ctx - konteks canvas, x,y,w,h - jak w JSON
var kappa = .5522848;
var ox = (w / 2) * kappa,
  oy = (h / 2) * kappa,
  xe = x + w,
  ye = y + h,
  xm = x + w / 2,
  ym = y + h / 2;
ctx.beginPath();
ctx.moveTo(x, ym);
ctx.bezierCurveTo(x, ym - oy, xm - ox, y, xm, y);
ctx.bezierCurveTo(xm + ox, y, xe, ym - oy, xe, ym);
ctx.bezierCurveTo(xe, ym + oy, xm + ox, ye, xm, ye);
ctx.bezierCurveTo(xm - ox, ye, x, ym + oy, x, ym);
ctx.closePath();
```

4.2.2 Kwadraty

```
virtual void QPaintEngine::drawRects( const QRectF * rects,
                                      int rectCount );
virtual void QPaintEngine::drawRects( const QRect * rects,
                                      int rectCount );
```

```
{
  "t":"rect",
  "x":0.0,       // Pozycja lewego-gornego wierzcholka X
  "y":0.0,       // Pozycja lewego-gornego wierzcholka Y
  "w":10.0,      // Szerokosc
  "h":10.0       // Wysokosc
}
```

Pełna implementacja, natywnie wspierane w *canvas* za pomocą metody *fillRect*.

4.2.3 Linie

```
virtual void QPaintEngine::drawLines( const QLineF * lines,
                                      int lineCount );
virtual void QPaintEngine::drawLines( const QLine * lines,
                                      int lineCount );
```

```
{
  "t":"line",
  "xs":0.0,     // Pozycja startowa X
  "ys":0.0,     // Pozycja startowa Y
  "xe":10.0,    // Pozycja koncowa X
  "ye":10.0     // Pozycja koncowa Y
}
```

Pełna implementacja, natywnie wspierane w *canvas* za pomocą metod *moveTo* oraz *lineTo*.

4.2.4 Obrazy

```
virtual void QPaintEngine::drawImage( const QRectF & rectangle,
                                      const QImage & image,
                                      const QRectF & sr,
                                      Qt::ImageConversionFlags flags =
                                          Qt::AutoColor );
virtual void QPaintEngine::drawPixmap( const QRectF & r,
                                       const QPixmap & pm,
                                       const QRectF & sr );
virtual void QPaintEngine::drawTiledPixmap( const QRectF & rect,
                                            const QPixmap & pixmap,
                                            const QPointF & p );
```

```
{
  "t":"image",
  "data":"Ja8SA9c72b71HDj8", // Identyfikator obrazu
  "x":0.0,                   // Pozycja X
  "y":0.0                    // Pozycja Y
}
```

Pełna implementacja, natywnie wspierane w *canvas* za pomocą metody drawImage.

4.2.5 Wielokąty

```
virtual void QPaintEngine::drawPolygon( const QPointF * points,
                                        int pointCount,
                                        PolygonDrawMode mode );
virtual void QPaintEngine::drawPolygon( const QPoint * points,
                                        int pointCount,
                                        PolygonDrawMode mode );
```

```
{
  "t":"polygon",
  "mode":0, // 0: QPaintEngine::OddEvenMode
            // 1: QPaintEngine::WindingMode
            // 2: QPaintEngine::ConvexMode
            // 3: QPaintEngine::PolylineMode
            // http://doc.qt.digia.com/stable/qpaintengine.html#
               PolygonDrawMode-enum
  "data":   // Lista punktow do polaczenia
    [
      [0.0,0.0],
      [10.0,10.0],
      [123.0,123.0]
    ]
}
```

Pełna implementacja, natywnie wspierane w *canvas* za pomocą metod *moveTo*, *lineTo* oraz *closePath*.

4.2.6 Punkty

```
virtual void QPaintEngine::drawPoints( const QPointF * points,
                                       int pointCount );
virtual void QPaintEngine::drawPoints( const QPoint * points,
                                       int pointCount );
```

```
{
  "t":"points",
  "data":              // Lista punktow
    [
      [0.0,0.0],
      [10.0,10.0],
      [123.0,123.0]
    ]
},
```

Pełna implementacja, natywnie wspierane w *canvas*, za pomocą *strokeRect* rysowany jest kwadrat o rozmiarach 1 na 1 piksel.

4.2.7 Ścieżki

```
virtual void QPaintEngine::drawPath( const QPainterPath & path );
```

```
{
  "t":"path",
  "data":          // Lista punktow
    [
```

```
      ["t":0,"p":[[0,0]]],       // moveTo
      ["t":1,"p":[[10,10]]],     // lineTo
      ["t":2,"p":[[10,10],[100,100]]],  // quadTo
      ["t":2,"p":[[10,10],[100,100],[1000,1000]]],  // cubicTo
    ],
  "fill":0 // 0: Qt::OddEvenFill
           // 1: Qt::WindingFill
           // http://doc.qt.digia.com/stable/qt.html#FillRule-enum
}
```

Pełna implementacja, natywnie wspierane w *canvas* za pomocą metod *lineTo*, *lineTo*, *quadraticCurveTo* oraz *bezierCurveTo*.

4.2.8 Tekst

```
virtual void QPaintEngine::drawTextItem( const QPointF & p,
                                         const QTextItem & textItem );

{
  "t":"text",
  "data":
  {
    "text":"Przykladowy tekst",     // Tekst w kodowaniu UTF8
    "ascent":0,                     // Dystans od linii bazowej do
                                    // najwyzej polozonego punktu
    "descent":0,                    // Dystans od linii bazowej do
                                    // najnizej polozonego punktu
    "x":0,                          // Pozycja X
    "y":0,                          // Pozycja Y
    "font":"CSS-format font string" // Informacje o czcionce
                                    // w formacie CSS
  }
}
```

Pełna implementacja, natywnie wspierane w *canvas*. Dokładny opis obsługi w podrozdziale dotyczącym pędzla.

4.2.9 Stan pędzli

```
virtual void QPaintEngine::updateState(const QPaintEngineState& state)

{
  "t":"state",
  "data":
    {
      // Opis pedzla krawedzi
      "pen": { /* ... */ },

      // Opis pedzla wypelnienia
      "brush": { /* ... */ },

      // Czcionka
      "font": "Opis czcionki w formacie CSS",
```

```
      // Offset pedzla
      "brushorigin":
        {
          "x":0.0,
          "y":0.0
        },

      // Macierz transformacji 3x3
      "transform":[[1,0,0],[0,1,0],[0,0,1]],

      // Metoda kompozycji
      "composition":"source-over",

      // Opis pedzla tla
      "bbrush": { /* ... */ },

      // Przezroczystosc
      "opacity":0.5,

      // Obcinanie
      "clip": { /* ... */ }
    }
}
```

Pędzele wypełnienia i tła

- Brak wypełnienia

  ```
  {
    "style":0,
  }
  ```

  Pełna implementacja za pomocą ustawienia koloru na RGB(0,0,0,0) – całkowicie przeźroczysty.

- Gradient liniowy

  ```
  {
    "style":15,
    "gradient":
      {
        "type":0,
        "xs":0.0, // Punkt poczatkowy X
        "ys":0.0, // Punkt poczatkowy Y
        "xe":0.0, // Punkt koncowy X
        "ye":0.0, // Punkt koncowy Y
        "stops":  // Lista kolorow (pary [odleglosc, kolor])
                  // Ogleglosc wzgledna z przedzialu (0;1)
          [
            [0,"#FFFFFF"],
            [1,"#000000"]
          ],
        "spread":0, // Wypelnienie poza obszarem gradientu
                    // 0: QGradient::PadSpread
                    // 1: QGradient::ReflectSpread
  ```

```
                    // 2: QGradient::RepeatSpread
        "mode":0    // Definiuje sposob interpretacji wspolrzednych
                    // 0: QGradient::LogicalMode
                    // 1: QGradient::StretchToDeviceMode
                    // 2: QGradient::ObjectBoundingMode
      }
    "transform":[[1,0,0],[0,1,0],[0,0,1]] // Opcjonalne
}
```

Pełna implementacja za pomocą *createLinearGradient*.

- Gradient kołowy

```
{
  "style":16,
  "gradient":
    {
      "type":1,
      "xc":0.0, // Punkt srodkowy X
      "yc":0.0, // Punkt srodkowy Y
      "xf":0.0, // Punkt koncowy X
      "yf":0.0, // Punkt koncowy Y
      "stops":  // Lista kolorow (pary [odleglosc, kolor])
                // Ogleglosc wzgledna z przedzialu (0;1)
        [
          [0,"#FFFFFF"],
          [1,"#000000"]
        ],
      "spread":0,
      "mode":0
    }
  "transform":[[1,0,0],[0,1,0],[0,0,1]] // Opcjonalne
}
```

Pełna implementacja za pomocą *createRadialGradient*.

- Gradient stożkowy

```
{
  "style":17,
  "gradient":
    {
      "type":2,
      "xc":0.0, // Punkt srodkowy X
      "yc":0.0, // Punkt srodkowy Y
      "a":0.0,  // Kat
      "stops":  // Lista kolorow (pary [odleglosc, kolor])
                // Ogleglosc wzgledna z przedzialu (0;1)
        [
          [0,"#FFFFFF"],
          [1,"#000000"]
        ],
      "spread":0,
      "mode":0
    }
  "transform":[[1,0,0],[0,1,0],[0,0,1]] // Opcjonalne
```

```
}
```

Brak implementacji w *canvas*.

- Tekstura

```
"brush":
  {
    "style":24,
    "image":"
       DIUSHFIUSHRIUDSHIFIUHI329859vdsy7vy87dv8sgv87sdgvgsd8gvyu)
       ",
    "transform":[[1,0,0],[0,1,0],[0,0,1]] // Opcjonalne
  }
```

Implementacja częściowa za pomocą *createPattern*, bez możliwości określenia transformacji.

- Kolor

```
{
  "style":others,
  "color":"#FFFFFF"
  [,"transform":[[1,0,0],[0,1,0],[0,0,1]]]
}
```

Pełna implementacja w *canvas*.

Pędzel krawędziowy

Istnieje kilka rodzajów pędzli krawędziowych. Każdy z nich posiada inną strukturę przesyłanych danych, które przedstawiono poniżej. W tym miejscu należy również zaznaczyć kilka opcji wspólnych, występujące we wszystkich rodzajach pędzli:

1. Zakończenia linii *cap*[1]

   - *Qt::FlatCap* (wartość 0x00) — proste ścięcie linią prostopadłą do stycznej na końcu krzywej,
   - *Qt::SquareCap* (wartość 0x10) — zakończenie kwadratowe, bardzo podobne do poprzedniego typu,
   - *Qt::RoundCap* (wartość 0x20) — gładkie, zaokrąglone zakończenie.

   Wszystkie opcje dostępne w *canvas*.

2. Złączenia (załamania) linii *join*[2]

   - *Qt::MiterJoin* (wartość 0x00) [3]

[1] http://doc.qt.digia.com/qt/qt.html#PenCapStyle-enum
[2] http://doc.qt.digia.com/qt/qt.html#PenJoinStyle-enum
[3] Wraz z tą wartością parametru musi dodatkowo pojawić się parametr *miter* określający promień zaokrąglania załamań linii.

- *Qt::BevelJoin* (wartość 0x40)
- *Qt::RoundJoin* (wartość 0x80)
- *Qt::SvgMiterJoin* (wartość 0x100)

Opcja *Qt::SvgMiterJoin* niedostępna w *canvas* z wiadomych względów.

3. Szerokość linii *width*

   Wartość tego parametru stanowi szerokość linii mierzoną w pikselach.

Pełne wsparcie w *canvas*.

W celu wyczyszczenia wartości pędzla jako wartość dla klucza *pen* wysyłany jest obiekt pusty.

- Kolor

```
"pen":
  {
    "color":"#FFFFFF",
    "cap":0,
    "join":1,
    "width":10
  }
```

  Pełne wsparcie w *canvas*.

- Linia przerywana

```
"pen":
  {
    "color":"#FFFFFF",
    "dash":   // !!! optional
      {
        "offset":0.5,
        "pattern":[10,10,30,10,20,10]
      },
    "cap":0,
    "join":1,
    "width":10
  }
```

  Brak wsparcia *canvas*. Możliwe jedynie symulowanie za pomocą pojedynczych kresek. Brak implementacji w projekcie.

- Linia z teksturą

```
"pen":
  {
    "brush": { /* Patrz opis pedzla wypelnienia */ },
    "cap":0,
    "join":0,
    "miter":0.5,
    "width":10
  }
```

  Pełne wsparcie w *canvas*.

Czcionka

Opis czcionki reprezentowany jest w formacie CSS[4] i obejmuje kolejno:

- Styl
- Wariant
- Rozmiar
- Wysokość
- Rodzinę (lista nazw rozdzielona przecinkami)

Przykład:

```
"font":"italic small-caps lighter 15px Sans-Serif"
```

Pełne wsparcie w *canvas*.

Metoda kompozycji

Parametr ten określa w jaki sposób łączone są kolejne nakładające się warstwy.[5]

- *source-atop*
- *source-in*
- *source-out*
- *source-over*
- *destination-atop*
- *destination-in*
- *destination-out*
- *destination-over*
- *lighter*
- *darker*
- *xor*
- *copy*

[4] http://www.w3schools.com/cssref/pr_font_font.asp
[5] http://doc.qt.digia.com/qt/qpainter.html#CompositionMode-enum

Lista mozliwych wariantów po stronie serwera jest zdecydowanie dłuższa od wyżej przedstawionej, która obejmujenie jedynie kolejność i sposoby łaczenia poszczególnych warstw ale również złożone metody mieszania kolorów. Serwer przesyła więc tylko te opcje, które są wspierane przez *canvas* po stronie klienta, a wszystkie pozostałe zastępuje domyślną wartością *source-atop*.

Obcinanie

```
"clip":
  {
     // Lista punktow sciezki ciecia
     "data":
       [
         ["t":0,"p":[[0,0]]],                              // moveTo
         ["t":1,"p":[[10,10]]],                            // lineTo
         ["t":2,"p":[[10,10],[100,100]]],                  // quadTo
         ["t":2,"p":[[10,10],[100,100],[1000,1000]]], // cubicTo
       ],
     "fill":0 // 0: Qt::OddEvenFill
              // 1: Qt::WindingFill
  }
```

Obcinanie polega na ograniczaniu obszaru rysowania poprzez dowolną krzywą. Pełne wsparcie w *canvas*, jednak z powodu błędów w implementacji popularnych przeglądarek (Google Chrome oraz Mozilla Firefox) nie została ona zaimplementowana. Problem został opisany w podrozdziale 4.6.

4.3 Zdarzenia po stronie klienta

Po stronie przeglądarki przechwytywane są wszystkie zdarzenia myszy oraz klawiatury. Każde zdarzenie jest zamieniane na obiekt *JSON* i wysyłane do serwera przy użyciu *Websocket*. Dodatkowo przesyłane są zdarzenia dotyczące manipulacji oknami – zdarzenia zmiany rozmiary, zamknięcia oraz aktywacji okna. Serwer na podstawie otrzymanych danych tworzy i wstawia zdarzenia do pętli zdarzeń (ang. event loop). W ten sposób z poziomu przeglądarki możliwa jest całkowita kontrola aplikacji, dla końcowego użytkownika równoważna funkcjonalnie z pracą na zdalnym urządzeniu.

W zdarzeniach myszy oraz klawiatury wszystkie przekazywane wartości są zgodne z wewnętrznym systemem zdarzeń Qt, przez co nie jest wymagana dodatkowa konwersja po stronie serwera. Wszystkie wartości pozycji, szerokości i wysokości określone są w pikselach.

W rozdziale przedstawiono przykłady przesyłanych obiektów *JSON* wraz z komentarzem.

4.3.1 Zdarzenia myszy

W zdarzeniach myszy używane są następujące pola:

- type – typ zdarzenia,
- id – identyfikator widgeta, którego dotyczy zdarzenie,
- x – pozycja na osi X w momencie zajścia zdarzenia,
- y – pozycja na osi Y w momencie zajścia zdarzenia,
- ox – poprzednia pozycja na osi X, używane tylko przy ruchu,
- oy – poprzednia pozycja na osi Y, używane tylko przy ruchu,
- btn – wartość liczbowa określająca przyciski wciśnięte w momencie zajścia zdarzenia,
- modifiers – wartość liczbowa określająca klawisze specjalne (Alt, Control, Shift, klawisz Windows, klawisz Menu) wciśnięte w momencie zajścia zdarzenia,
- delta – wartość przesunięcia, używane tylko przy przewijaniu,
- orientation – kierunek, używane tylko przy przewijaniu.

Ruch myszy

```
{
  "command": "mouse",
  "type": "move",
  "id": 123456,  // Identyfikator obiektu, ktorego dotyczy zdarzenie
  "x": 0.0,
  "y": 0.0,
  "ox": 0.0,
  "oy": 0.0,
  "btn": 0x0,     // 0x00000000 Qt::NoButton
                  // 0x00000001 Qt::LeftButton
                  // 0x00000002 Qt::RightButton
                  // 0x00000004 Qt::MiddleButton
                  // 0x00000008 Qt::XButton1
                  // 0x00000010 Qt::XButton2
  "modifiers": 0x0  // 0x00000000 Qt::NoModifier
                    // 0x02000000 Qt::ShiftModifier
                    // 0x04000000 Qt::ControlModifier
                    // 0x08000000 Qt::AltModifier
                    // 0x10000000 Qt::MetaModifier
                    // 0x20000000 Qt::KeypadModifier
                    // 0x40000000 Qt::GroupSwitchModifier
}
```

Wciśnięcie przycisku myszy

```
{
  "command": "mouse",
  "type": "press",
  "id": 123456,
```

```
  "x": 0.0,
  "y": 0.0,
  "btn": 0x00000001,        // Qt::LeftButton
  "modifiers": 0x00000000   //  Qt::NoModifier
}
```

Zwolnienie przycisku myszy

```
{
  "command": "mouse",
  "type": "release",
  "id": 123456,
  "x": 0.0,
  "y": 0.0,
  "btn": 0x00000001,        // Qt::LeftButton
  "modifiers": 0x02000000   //  Qt::ShiftModifier
}
```

Podwójne kliknięcie

```
{
  "command": "mouse",
  "type": "dbclick",
  "id": 123456,
  "x": 0.0,
  "y": 0.0,
  "btn": 0x00000002,        // Qt::RightButton
  "modifiers": 0x00000000   //  Qt::NoModifier
}
```

Zmiana położenia kółka myszy

```
{
  "command": "wheel",
  "id": 123456,
  "x": 0.0,
  "y": 0.0,
  "btn": 0x00000002,        // Qt::RightButton
  "modifiers": 0x00000000   //  Qt::NoModifier
  "delta": 120,          // Zmiana polozenia
  "orientation": 0x      // 0x1 Qt::Horizontal
                         // 0x2 Qt::Vertical
}
```

4.3.2 Zdarzenia klawiatury

W zdarzeniach klawiatury używane są następujące pola:

- type – typ zdarzenia,
- key – kod klawisza
- text – ciąg znaków będący rezultatem zdarzenia

- autorep – wartość logiczna określająca, czy zdarzenie jest wynikiem przytrzymania klawisza
- count – ilość powtórzeń
- modifiers – analogicznie jak w przypadku zdarzeń myszy

Wciśnięcie klawisza

```
{
  "command": "key",
  "type": "press",
  "key": 0x193,     // Kod klawisza
  "text": "a",      // Ciag znakow UTF8 bedacy rezultatem zdarzenia
  "autorep": true|false, // Okresla czy zdarzenie jest wynikiem
                         // przytrzymania klawisza przed dluzszy czas
  "count": 0,     // Okresla ile powtorzen klawisza mialo miejsce
  "modifiers": 0x0  // 0x00000000 Qt::NoModifier
                    // 0x02000000 Qt::ShiftModifier
                    // 0x04000000 Qt::ControlModifier
                    // 0x08000000 Qt::AltModifier
                    // 0x10000000 Qt::MetaModifier
                    // 0x20000000 Qt::KeypadModifier
                    // 0x40000000 Qt::GroupSwitchModifier
}
```

Zwolnienie klawisza

```
{
  "command": "key",
  "type": "release",
  "key": 0x193,     // Kod klawisza
  "text": "a",      // Ciag znakow UTF8 bedacy rezultatem zdarzenia
  "autorep": true|false, // Okresla czy zdarzenie jest wynikiem
                         // przytrzymania klawisza przed dluzszy czas
  "count": 0,     // Okresla ile powtorzen klawisza mialo miejsce
  "modifiers": 0x0  // Qt::NoModifier
}
```

4.3.3 Zdarzenia okien

W zdarzeniach okien używane są następujące pola:

- id – tak jak w przypadku eventów myszy liczba będąca identyfikatorem widgetu, którego dotyczy zdarzenie
- w – wysokość
- h – szerokość

Zmiana rozmiaru okna

```
{
  "command": "resize",
  "id": 123456,    // Identyfikator obiektu, ktorego dotyczy zdarzenie
  "w": 100,        // Nowa szerokosc
  "h": 200         // Nowa wysokosc
}
```

Aktywacja okna

```
{
  "command": "activate",
  "id": 123456   // Identyfikator obiektu, ktorego dotyczy zdarzenie
}
```

Zamknięcie okna

```
{
  "command": "close",
  "id": 123456   // Identyfikator obiektu, ktorego dotyczy zdarzenie
}
```

4.4 Szczegóły implementacji po stronie serwera

4.5 Szczegóły implementacji po stronie klienta

4.5.1 Biblioteki pomocnicze

Podczas pracy z projektem zostały wykorzystane popularne biblioteki ułatwiające podstawowe zadania. Wszystkie z nich są darmowe nawet w wykorzystaniu komercyjnym oraz całkowicie otwarte.

1. jQuery — lekka biblioteka ułatwijąca operacje na elementach DOM i ułatwiająca pracę z językiem Javascript
2. jQuery-UI — plugin do biblioteki jQuery umożliwiający tworzenie zaawansowanych wizualnych elementów, takich jak okna dialogowe, elementy rozszerzalne (ang. resizable), elementy przesuwalne (ang. draggable).
3. jQuery-mousewheel — plugin do biblioteki jQuery dodający obsługę kółka myszy
4. sylvester.js — biblioteka służąca do zaawansowanych obliczeń na macierzach i wektorach

4.5.2 Połączenie

Klient po załadowaniu początkowej strony dokonuje połączenia WebSocket z serwerem na port podany na stronie. Następnie nasłuchuje na informacje od serwera i na każdą z nich reaguje. Komunikaty w formacie JSON w drugą stronę wysyłane są po zajściu zdarzeń po stronie klienta, np. ruchu myszą. Dokładny opis zdarzeń znajduję się w sekcji 4.3.

4.5.3 Window manager (ang. zarządca okien)

Typowe programy komputerowe składaja się z wielu okien. Sposób implementacji pseudookien i zarządcy w projekcie był możliwy na dwa sposoby. Pierwszym wariantem jest zastosowanie osobnych okien przeglądarki (tzw. popupów), które odpowiadałyby rzeczywistym oknom przeglądarki. Drugą opcją jest stworzenie minimalistycznego managera okien w języku Javascript.

Minusem pierwszym rozwiązania jest całkowitym brak możliwości sterowania oknami. Przeglądarka nie jest w stanie zablokować możliwości zamknięcia okna, sterować ich modalnościa oraz przyciskami sterowania (minimalizacji, maksymalizacji, zamknięcia i innymi). Co więcej stosowanie dodatkowych okien przeglądarki jest uważane za złą praktykę.

Z powodu tak dużej ilości problemów w projekcie zdecydowano się na użycie drugiego wariantu. Do stworzenia okien wykorzystana została biblioteka jQuery-UI dostarczająca metody umożliwiający w przystępny sposób tworzenie elementów rozszerzalnych (ang. resizable) oraz przeciągalnych (ang. draggable). Celem funkcjonalnym było upodobnienie zachowania pseudookien w przeglądarce do prawdziwych okien programu:

- zmiana rozmiaru okna przy pomocy uchwytów w rogach i na krawędziach,
- przenoszenie okien za pomocą paska tytułowego,
- minimalizacja, maksymalizacja oraz zamykanie okien za pomocą przycisków w prawym górnym rogu.

TODO - tutaj rysunk okienka z zaznaczonymi elementami typu pasek Bar, przyciski -,[],X, chywataki resizable.

Okno na stronie jest elementem blokowym typu *div* z zagnieżdżonymi elementami symulującymi swoje odpowiedniki z systemowego menedżera okien.

Zaimplementowana funkcjonalność maksymalizacji okna po stronie przeglądarki różni się od funkcjonalności w rzeczywistym środowisku (po stronie serwera). Maksymalizacja polega na zwiększeniu rozmiarów okna do maksymalnego dostępnu obszaru na stronie HTML. Takie rozwiązanie wynika z możliwości uruchomienia serwera aplikacji w serwerze X11 w dowolnej rozdzielczości. Po faktycznym zmaksymalizowaniu okna

na serwerze użytkownik po stronie przeglądarki widziałby okno o rozmiarze innym niż pełny dostępny obszar widoku w przeglądarce. W przypadku rozmiaru mniejszego skutkowałoby to pustym, niezagospodarowanym miejscem w oknie przeglądarki, natomiast większy rozmiar powodowałby pojawienie się suwaków przewijania (ang. scrollbar).

Funkcjonalność minimalizacji okna jest również symulowana. Okno jest chowane poprzez zmianę wartości CSS *display* na *none*. Dodatkowo tworzony jest element na pasku zadań, który po kliknięciu przywraca ukryte okno. Pasek zadań jest elementem strony znajdującym się na samym dole.

** TODO rysunek paska zadań **

4.5.4 Rysowanie pojedynczego widgeta

Widget reprezentowany jest przez element *div* zawierający dwa elementy: element *canvas* oraz kontener *div* na widgety potomne posiadające analogiczną struktruę. Element ten ma pełni funkcję pomocniczą (jest w pełni przeźroczysty) i służy do umiejscowienia elementu w stosounku do jego rodzica za pomocą atrybutów CSS *left* oraz *top*. Taka implementacja relacji rodzic-dziecko widgetów na stronie zapewnia drzewiastość i łatwość zarządzania widgetami. Zmiana rodzica widgeta, który posiada zagnieżdzone dzieci nie jest problematyczna, a ze względu na format przesyłanych danych od serwera ta operacja jest bardzo często używana na etapie tworzenia okien. Widgety z flagą *Qt::Window*, czyli przede wszystkim dziedziczące z *QDialog* oraz *QMainWindow* są traktowane w specjalny sposób. Są opakowywane w kontener okna opisany powyżej i są elementami stojącymi najwyżej w strukturze drzewa.

4.6 Napotkane problemy

1. Rysowanie za pomocą zdarzeń, synchronizacja i buforowanie
2. Znikający *focus* okna aplikacji
3. Kody znaków klawiatury
4. ....
5. problem z clippingiem w przeglądarkach

Ad.1 Rysowanie widgetów we frameworku *Qt* realizowane jest wewnątrz kolejki zdarzeń aplikacji. Zdarzenie rysowania powiadamia element interfejsu o konieczności przerysowania. Widget posiada wskaźnik do miejsca w pamięci gdzie powinien przeprowadzic operację renderowania, gdyż sam jest implementacją klasy QPaintDevice.

Powyższy schemat działania wymagał znalezienia sposobu na zmuszenie widgetów do rysowania za pomocą specjalnie przygotowanej implementacji klasy QPaintEngine oraz QPaintDevice. Do tego celu wykorzystana została metoda biblioteki Qt:

```
void QWidget::render(QPainter * painter,
                     const QPoint & targetOffset = QPoint(),
                     const QRegion & sourceRegion = QRegion(),
                     RenderFlags renderFlags
                          = RenderFlags(DrawWindowBackground |
                                        DrawChildren))
```

Umożliwiła nam ona wskazanie obiektu QPainter wykorzystującego mechanizm renderowania serwera, tj. reimplementacje klasy QPaintEngine oraz QPaintDevice. Wykorzystanie tej metody powoduje wygenerowanie kolejnego zdarzenia i umieszczenie go w kolejce aplikacji. Powodowało to problem wpadania serwera w nieskończoną pętlę i uniemożliwiało jego dalsze poprawne funkcjonowanie.

Rozwiązaniem problemu było stworzenie własnej kolejki widgetów, które wymagają renderowania. Kolejka ta jest opróżniana w pewnych odstępach czasu nie krótszych niż 100 milisekund. Wartość ta została dobrana eksperymentalnie tak aby uzyskać efekt płynnej interakcji z aplikacją.

Ad.2 Biblioteka *Qt* stanowi niejako nakładkę dla natywnego zarządcy okien systemu operacyjnego. W zwizku z tym o kolejności okien na stosie decyduje system operacyjny. *Qt* podejmuje jedynie odpowiednie czynności w celu aktualizacji graficznego interfejsu aplikacji w zależności od aktualnego stanu konkretnych okien definiowanego przez system operacyjny. W sytuacji kiedy użytkownik nie prowadzi żadnej intrakcji z systemem operacyjnym a jedynie z aplikacją, mogło by się zdarzyć, że niektóe zdarzenia było by ignorowane przez aplikację. Przykładem takiej sytuacji jest wprowadzanie tekstu na klawiaturze. *Qt* przesyła zdarzenia klawiatury do widgeta, który aktualnie posiada focus w oknie, które znajduje się na szczycie stosu okien w systemie operacyjnym. W momencie kiedy dwóch zdalnych użytkowników uruchomiło by aplikację na tym samym serwerze, zdarzenia jednego użytkownika były by ignorowane ponieważ tylko jedno okno jednej aplikacji może być na szczycie stosu.

Rozwiązaniem tego problemu była reimplementacja odpowiednich metod *Qt* w celu symulacji zachowania stosu okien systemu operacyjnego wewnątrz samej aplikacji. W rezultacie aplikacja działa tak jakby zawsze była aktywna, dzięki czemu framework *Qt* poprawnie renderuje wszystkie elementy interfejsu użytkownika.

Ad.3 *Qt* dostarcza platformowo niezależny opis kodów znaków klawiatury za pomocą zdefiniowanego typu wyliczeniowego *Qt::Key*[6]. Niestety przeglądarki nie są dobrze ustandaryzowane i ich numeracja klawiszy znacznie różni się nie tylko między samymi platformami ale również między ich wersjami. Dodatkowo przeglądarki często nie wspierają wszystkich klawiszy przez to zakres kodów znaków jest inny niż w przypadku biblioteki *Qt*.

Powyższe problemy wymusiły konieczność zaimplementowania metody konwertującej po stronie klienta kody klawiszy na odpowiadające im wartości typu wyliczeniowego *Qt::Key*.

—— TODO: Jak to Janek zrobił

[6]http://doc.qt.digia.com/qt/qt.html#Key-enum

Rozdział 5

Testy aplikacji

...

5.1 Testy w środowisku lokalnym

...

5.2 Testy w sieci Internet

...

Rozdział 6

Podsumowanie

Temat pracy inżynierskiej został w pełni zrealizowany, a jej wynikiem jest prototyp serwera oraz webowej aplikacji klienckiej.

Program serwera udostępnia kod strony WWW, który uruchamiany jest przez przeglądarkę. Uruchamia on również aplikację graficzną opartą o framework Qt oraz odpowiada za utworzenie połączenia między klientem a procesem aplikacji. Możliwa jest także obsługa wielu połączeń równocześnie. Przy pomocy techniki wstrzykiwania kodu bibliotek linkowanych dynamicznie *(ang. DLL injection)* oraz wewnętrznych mechanizmów biblioteki *Qt*, użycie aplikacji nie wymaga ponownej kompilacji, zarówno samego frameworka *Qt*, jak i uruchamianych aplikacji użytkowych.

Aplikacja kliencka stworzona w postaci dynamicznej strony WWW pokrywa bardzo duży podzbiór funkcjonalności modułu graficznych interfejsów biblioteki *Qt*. Wszelkie braki wynikają z niedoskonałości standardu HTML5, który wciąż jest mocno rozwijany.

Projekt będzie kontynuowany w następujących kierunkach:

- umożliwienie współpracy z aplikacjami opartymi o najnowszą bibliotekę *Qt* w wersji *5.0*,
- rozwinięcie zabezpieczeń — autentykacja i autoryzacja klientów,
- stworzenie panelu administracyjnego serwera oraz nowego widoku głównego aplikacji,
- automatyzacja procesu instalacji,
- utworzenie wersji serwera dla systemów *Windows* oraz *Mac OS*,
- rozwinięcie możliwości aplikacji klienckiej przy wykorzystaniu elementów technologii *HTML5*, które będą dostępne w przyszłości.