Zachodniopomorski Uniwersytet Technologiczny
Wydział Elektryczny
Katedra Sterowania i Pomiarów

SPRAWOZDANIE Z WYKONANIA ĆWICZENIA PROJEKTOWEGO Z PRZEDMIOTU STEROWANIE PROCESAMI DYSKRETNYMI

- **Kierunek**: Automatyka i Robotyka
- **Prowadzący**: dr inż. Maja Kocoń
- **Temat**: Algorytm najkrótszej drogi dla pojazdu w dwuwymiarowym otoczeniu.
- **Data oddania projektu**: 29.12.2017r.
- **Wykonał:** Marcin Nowosad, nr albumu 20139

Wstęp

Celem projektu było opracowanie algorytmu znajdowania najkrótszej drogi dla pojazdu poruszającego się w przestrzeni 2D. Pojazd ten jest modelem czołgu, sterowanym przez sztuczną inteligencję w grze komputerowej OpenFire[1].

Środowisko, w którym porusza się pojazd zawiera przeszkody w postaci nieprzejezdnych akwenów oraz zniszczalnych murów i zabudowań, a także sterowanych przez sztuczną inteligencję działek obronnych.

Celem AI pojazdu jest znalezienie takiej drogi, która będzie możliwie krótka, jednocześnie unikając kontaktu i niszczenia wrogich wieżyczek i/lub murów, o ile znacząco nie wydłuży to drogi.

Z racji faktu, że gra OpenFire dedykowana jest komputerom klasy Amiga, dodatkowym wymaganiem jest wysoka wydajność obliczeniowa opracowanego algorytmu. Ze względu na ograniczony rozmiar stosu, algorytm nie może być zdefiniowany w sposób rekurencyjny. Jeżeli jego czas obliczeń na procesorze MC68000 taktowanym 7MHz będzie dłuższy niż 1ms, powinien on zostać napisany w takiej formie, by móc go rozłożyć w czasie na kilka wywołań funkcji.

Skrót zasad gry

Każda drużyna dysponuje ograniczoną liczbą czołgów. Czołgi mogą poruszać się w 128 kierunkach ze skokiem kierunku co 2,81°.

Plansza składa się z baz zawierających punkty startowe, strzeżonych przez automatyczne wieżyczki strzelnicze. Każda baza ma punkt kontrolny, który pozwala drużynom na przejęcie bazy i uzyskanie przewagi terytorialnej. Wraz z przejęciem bazy przejmowane są strzegące jej wieżyczki. Dodatkowo gracze zyskują dostęp do znajdujących się weń punktów startowych, pozwalając im w razie zniszczenia pojazdu na szybszy powrót na przesuwający się front.

Przejęcie bazy odbywa się poprzez przebywanie pojazdów w pobliżu punktu kontrolnego – po pewnym czasie baza staje się ziemią neutralną, co wyłącza wieżyczki i punkty startowe, zaś po dłuższej chwili przechodzi ona w ręce drużyny docelowej. Przejmowanie bazy przyspiesza jeśli przy punkcie kontrolnym znajduje się więcej pojazdów danej drużyny. Tak samo przejęcie może zostać zahamowane lub odwrócone, jeśli broniąca drużyna będzie posiadać w zasięgu punktu kontrolnego odpowiednio: tyle samo lub więcej pojazdów, co przeciwnik.

Drużyny dysponują liczbą punktów, które ubywają:

- wraz ze stratą każdego z własnych pojazdów,
- wraz z upływem czasu, jeśli przeciwnik ma w swoim posiadaniu więcej punktów kontrolnych. Im większa dysproporcja tym szybciej ubywa punktów.

Drużyna przegrywa, gdy jej liczba punktów zmaleje do zera.

1 GitHub: https://github.com/tehKaiN/openFire

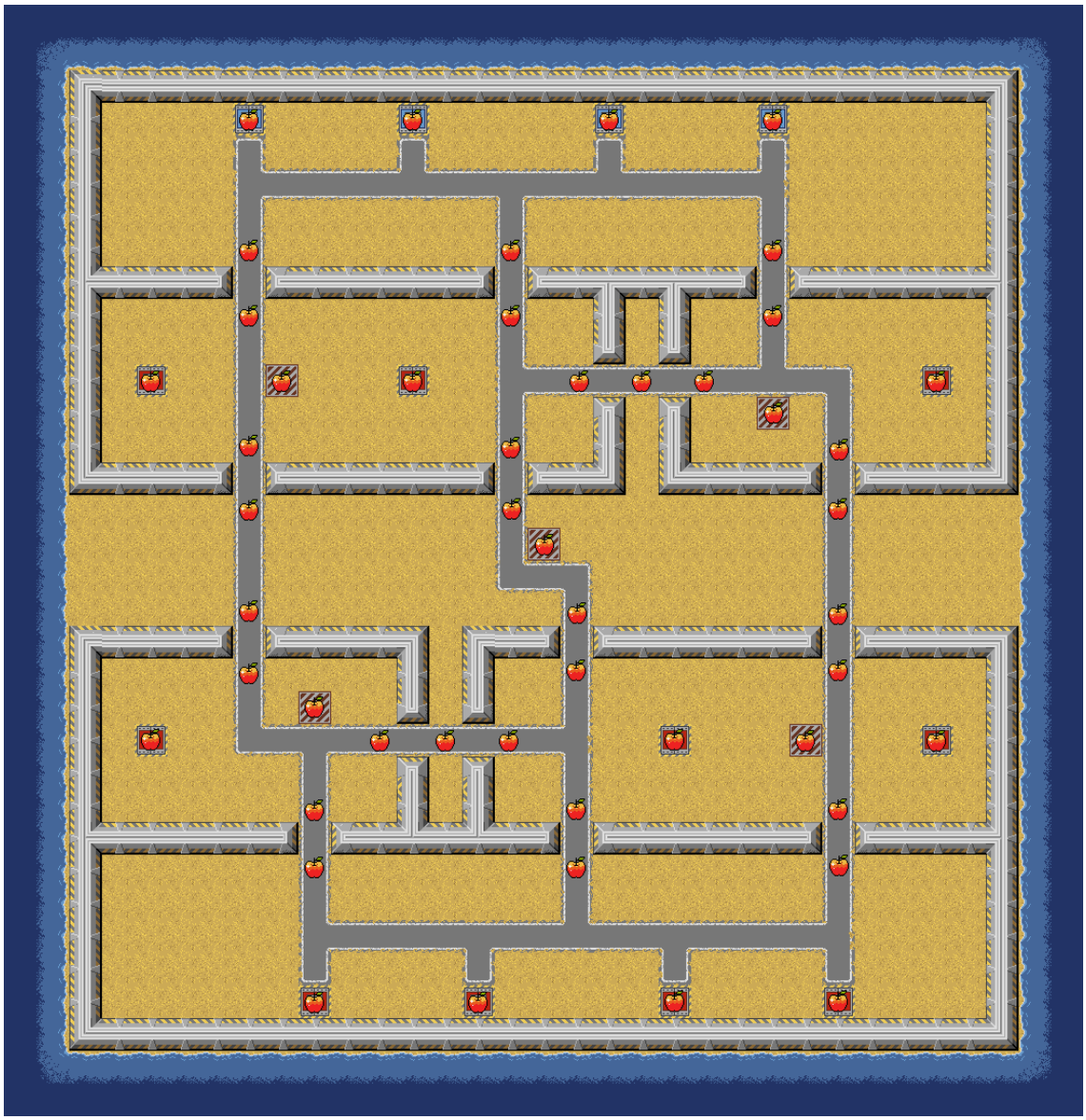
Przygotowanie danych do nawigacji

Węzły

Teren gry złożony jest z kafelków, każdy w rozmiarze 32x32 pikseli. Punkty nawigacyjne znajdują się na środku wybranych kafli. Aby zmniejszyć czas obliczeń opracowano algorytm, który umieszcza punkty nawigacyjne tylko w strategicznych miejscach:

- W punktach przejmowania,
- W punktach startowych,
- Przed i za przejazdami przez mur

Tym sposobem, na testowej mapie składającej się z 33x34 kafli, punktów nawigacyjnych jest tylko 49, co stanowi mniej niż 5% z liczby wszystkich pól. Poniżej przedstawiono testową mapę wraz z naniesionymi węzłami pod postacią jabłek.



Ryc. 1. Mapa z gry OpenFire. Węzły nawigacyjne zaznaczono ikoną jabłka.
Źródło: opracowanie własne, GIMP 2.8.

Koszty przejazdu przez kafle

Trasowanie powinno wybierać nie tylko drogę możliwie najkrótszą, lecz także w miarę bezpieczną. Droga nie powinna być sztucznie okrężna, dlatego też w granicach rozsądku algorytm powinien uwzględniać możliwość przejazdu przez przeszkody, w międzyczasie je niszcząc.

Dla każdego kafelka mapy obliczono koszt przejazdu:

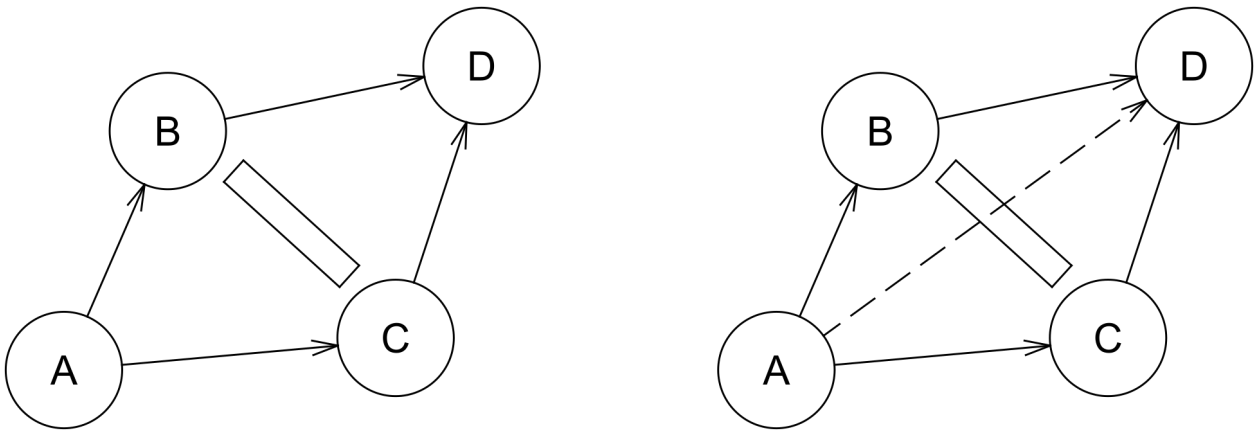
- Jeżeli jest to zwykły teren, koszt wynosi 1,
- Jeżeli na terenie znajduje się zniszczalny mur, koszt wynosi 100,
- Jeżeli pole jest wodą lub inną nieprzejezdną przeszkodą, koszt wynosi 255.
- Dodatkowo, koszt przejazdu rośnie o 5 za każdą wrogą wieżyczkę znajdującą się w zasięgu strzału.

Ze względu na udział wrogich wieżyczek strzelniczych w liczeniu kosztów przejazdu, dane te są przeliczane dwukrotnie: raz dla drużyny niebieskich, raz dla czerwonych.

Koszty przejazdu między węzłami

Zakłada się, że pojazdy będą przejeżdżać między węzłami nawigacyjnymi w linii prostej. Z tego względu warto zawczasu obliczyć sumaryczny koszt przejazdu na każdym z możliwych odcinków.

W sytuacji, gdzie ściany są niezniszczalne, połączenia prowadzone są tylko między węzłem a jego sąsiadami. W przypadku gdy teren jest zniszczalny, sąsiadami są wszystkie pozostałe węzły.

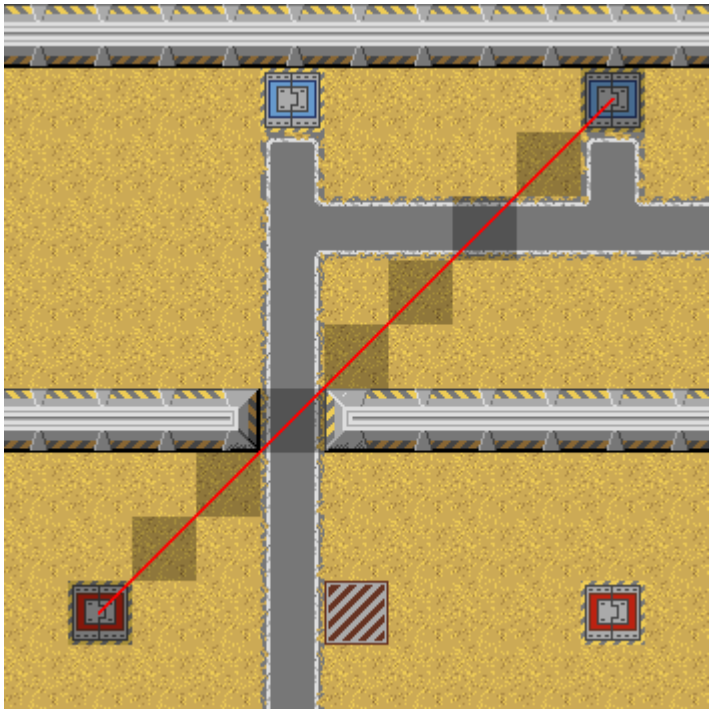


Ryc. 2. Porównanie nawigacji po terenie z przeszkodami niezniszczalnymi oraz zniszczalnymi. Źródło: opracowanie własne, DraftSight 2017.

Dane te trzymane są w kwadratowej tablicy, gdzie pierwszym indeksem jest numer węzła „z”, zaś drugim numer węzła „do”. Łatwo zauważyć, że tablica taka jest symetryczna.

Podejście naiwne

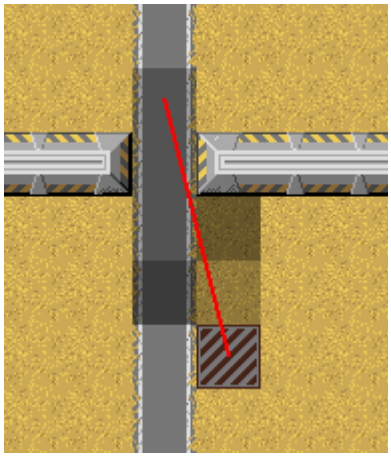
Naiwne tyczenie linii prostych między węzłami powoduje częste pomijanie kafli, a zatem zaniżanie kosztów na trasach. Przykładową sytuację zaniżenia kosztu przedstawiono poniżej.



Ryc. 3. Przykład naiwnego tyczenia kosztu drogi między węzłami w linii prostej.
Źródło: opracowanie własne, GIMP 2.8.

Zachowanie ciągłości linii

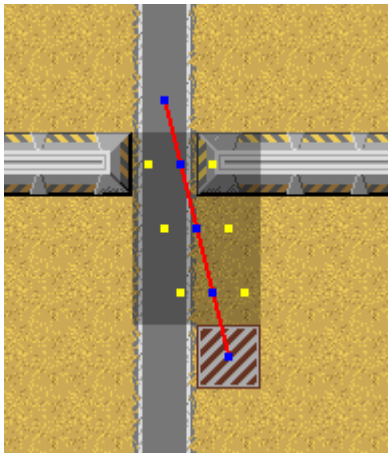
Kolejnym podejściem było wzbogacenie algorytmu rysowania linii tak, by dołączał do linii okoliczne kafle przy każdym przełamaniu linii. Podejście to jednak nie uwzględnia grubości pojazdu i faktu, że może on zahaczyć o okoliczne przeszkody.



Ryc. 4. Przykładowy problem przy tyczeniu kosztu drogi linią prostą z zachowaniem przejść w 4 kierunkach.
Źródło: opracowanie własne, GIMP 2.8.

Linie uwzględniające grubość pojazdu

Ostatecznie zdecydowano się na algorytm rysowania linii, który z każdym krokiem odsuwa dwa punkty o grubość pojazdu i dodaje koszt z kafli, na których te punkty leżą.



Ryc. 5. Tyczenie kosztu drogi między węzłami z uwzględnieniem szerokości pojazdu. Źródło: opracowanie własne, GIMP 2.8.

Ostateczny algorytm oparty o DDA przedstawiono poniżej:

```
UWORD aiCalcCostBetweenNodes(tAiNode *pFrom, tAiNode *pTo) {
 BYTE bDeltaX = pTo->fubX - pFrom->fubX; // Liczba kafli między źródłem a celem na osi X
 BYTE bDeltaY = pTo->fubY - pFrom->fubY; // oraz na osi Y.
 if(!bDeltaX && !bDeltaY) // Jeśli źródło = cel, to koszt jest zerowy
  return 0;
 // Dokładna pozycja punktu startowego - przejście ze współrzędnych kafli na piksele
 const fix16_t fHalf = fix16_one>>1;
 fix16_t fFineX = fix16_from_int((pFrom->fubX << MAP_TILE_SIZE) + MAP_HALF_TILE) + fHalf;
 fix16_t fFineY = fix16_from_int((pFrom->fubY << MAP_TILE_SIZE) + MAP_HALF_TILE) + fHalf;
 UBYTE ubAngle = getAngleBetweenPoints( // Pomiar kąta między źródłem a celem
  pFrom->fubX << MAP_TILE_SIZE, pFrom->fubY << MAP_TILE_SIZE,
  pTo->fubX << MAP_TILE_SIZE, pTo->fubY << MAP_TILE_SIZE
 );
 tBCoordYX sPtA = { // Wektor odsunięcia punktu o 10 pikseli w prawo
  .bX = fix16_to_int(10 * csin(ubAngle)),
  .bY = fix16_to_int(10 * ccos(ubAngle))
 };
 tBCoordYX sPtB = { // Wektor odsunięcia punktu o 10 pikseli w lewo
  .bX = fix16_to_int(-10 * csin(ubAngle)),
  .bY = fix16_to_int(-10 * ccos(ubAngle))
 };
 FUBYTE fubStart, fubStop;
 fix16_t fDx, fDy;
 if(ABS(bDeltaX) > ABS(bDeltaY)) { // Prosta jest bardziej pozioma niż pionowa
  fDx = fix16_from_int(SGN(bDeltaX)*MAP_FULL_TILE); // Skok na X co kafel
  // Podziel Y na tyle części, ile kafli zajmie X
  fDy = fix16_from_int(bDeltaY*MAP_FULL_TILE) / ABS(bDeltaX);
  fubStart = MIN(pFrom->fubX, pTo->fubX); // Zakres pętli - od lewej do prawej
  fubStop = MAX(pFrom->fubX, pTo->fubX);
 }
```

```
 else { // Prosta jest bardziej pionowa niż pozioma
  // Podziel X na tyle części, ile kafli zajmie Y
  fDx = fix16_from_int(bDeltaX*MAP_FULL_TILE) / ABS(bDeltaY);
  fDy = fix16_from_int(SGN(bDeltaY)*MAP_FULL_TILE); // Skok na Y co kafel
  fubStart = MIN(pFrom->fubY, pTo->fubY); // Zakres pętli – od góry do dołu
  fubStop = MAX(pFrom->fubY, pTo->fubY);
 }
 UWORD uwCost = 0;
 for(FUBYTE i = fubStart+1; i != fubStop; ++i) { // Chodzenie po linii
  // Krok w przód
  fFineX += fDx;
  fFineY += fDy;

  // Odsuń się o 10 pikseli w prawo i dodaj koszt z kafla
  FUBYTE fubChkAX = (fix16_to_int(fFineX) + sPtA.bX) >> MAP_TILE_SIZE;
  FUBYTE fubChkAY = (fix16_to_int(fFineY) + sPtA.bY) >> MAP_TILE_SIZE;
  uwCost += s_pTileCosts[fubChkAX][fubChkAY];

  // Odsuń się o 10 pikseli w prawo, jeśli jest to inny kafel niż z pktu A to dodaj koszt
  FUBYTE fubChkBX = (fix16_to_int(fFineX) + sPtB.bX) >> MAP_TILE_SIZE;
  FUBYTE fubChkBY = (fix16_to_int(fFineY) + sPtB.bY) >> MAP_TILE_SIZE;
  if(fubChkBX != fubChkAX || fubChkBY != fubChkAY)
   uwCost += s_pTileCosts[fubChkBX][fubChkBY];
 }
 return uwCost;
}
```

Algorytmy znajdowania najkrótszej drogi

Na potrzeby projektu przeanalizowano algorytmy: Dijkstry oraz A*, będące stosowanymi od wielu lat rozwiązaniami w tej dziedzinie i wielokrotnie omówionymi w różnorakiej literaturze. Implementacje na rzecz projektu zostały opracowane na podstawie artykułu ze strony Red Blob Games[2].

Algorytm Dijkstry

Algorytm Dijkstry sprawdza koszt drogi między sąsiadami a obecnym położeniem. Jeśli koszt drogi z obecnego do sąsiada jest niższy niż z innych węzłów to zapamiętaj obecny węzeł jako poprzednika.

Kolejne sprawdzane węzły dodawane są do priorytetowej kolejki z priorytetem równym dotychczasowemu kosztowi drogi. Po sprawdzeniu wszystkich sąsiadów, kolejny węzeł z kolejki brany jest jako bazowy i czynności te są powtarzane. Algorytm kończy pracę gdy dotrze do węzła końcowego – drogę odczytuje się sprawdzając kolejnych poprzedników, począwszy od węzła końcowego.

Kod algorytmu przedstawiono poniżej.

2 Introduction to A*: https://www.redblobgames.com/pathfinding/a-star/introduction.html

```
void dijkstra(tRoute *pRoute, tAiNode *pSrcNode, tAiNode *pDstNode) {
 tHeap *pFrontier = heapCreate(AI_MAX_NODES*AI_MAX_NODES);
 tAiNode *pCameFrom[AI_MAX_NODES] = {0};
 UWORD pCostSoFar[AI_MAX_NODES];
 memset(pCostSoFar, 0xFF, sizeof(UWORD)*AI_MAX_NODES);

 pCameFrom[pSrcNode - g_pNodes] = 0;
 pCostSoFar[pSrcNode - g_pNodes] = 0;

 heapPush(pFrontier, pSrcNode, 0);

 while(pFrontier->uwCount) {
  tAiNode *pCurrNode = heapPop(pFrontier);
  if(pCurrNode == pDstNode)
   break;

  for(UWORD i = 0; i <= g_fubNodeCount; ++i) {
   tAiNode *pNextNode = &g_pNodes[i];
   if(pNextNode == pCurrNode)
    continue;

   UWORD uwCost = pCostSoFar[pCurrNode – g_pNodes]
     + aiGetCostBetweenNodes(pCurrNode, pNextNode);
   if(uwCost < pCostSoFar[pNextNode - g_pNodes]) {
    pCostSoFar[pNextNode - g_pNodes] = uwCost;
    UWORD uwPriority = uwCost;
    heapPush(pFrontier, pNextNode, uwPriority);
    pCameFrom[pNextNode - g_pNodes] = pCurrNode;
   }
  }
 }

 pRoute->uwCost = pCostSoFar[pDstNode - g_pNodes];
 pRoute->pNodes[0] = pDstNode;
 pRoute->ubNodeCount = 1;
 tAiNode *pPrev = pCameFrom[pDstNode - g_pNodes];
 while(pPrev) {
  pRoute->pNodes[pRoute->ubNodeCount] = pPrev;
  ++pRoute->ubNodeCount;
  pPrev = pCameFrom[pPrev - g_pNodes];
 }
 pRoute->ubCurrNode = pRoute->ubNodeCount-1;

 heapDestroy(pFrontier);
}
```

Algorytm A*

Algorytm A* jest delikatnie zmodyfikowaną wersją algorytmu Dijkstry – kolejkuje on kolejne pola do odwiedzenia przy użyciu dodatkowej funkcji heurystyki, która ma na celu szybsze nakierowanie algorytmu w stronę rozwiązania.

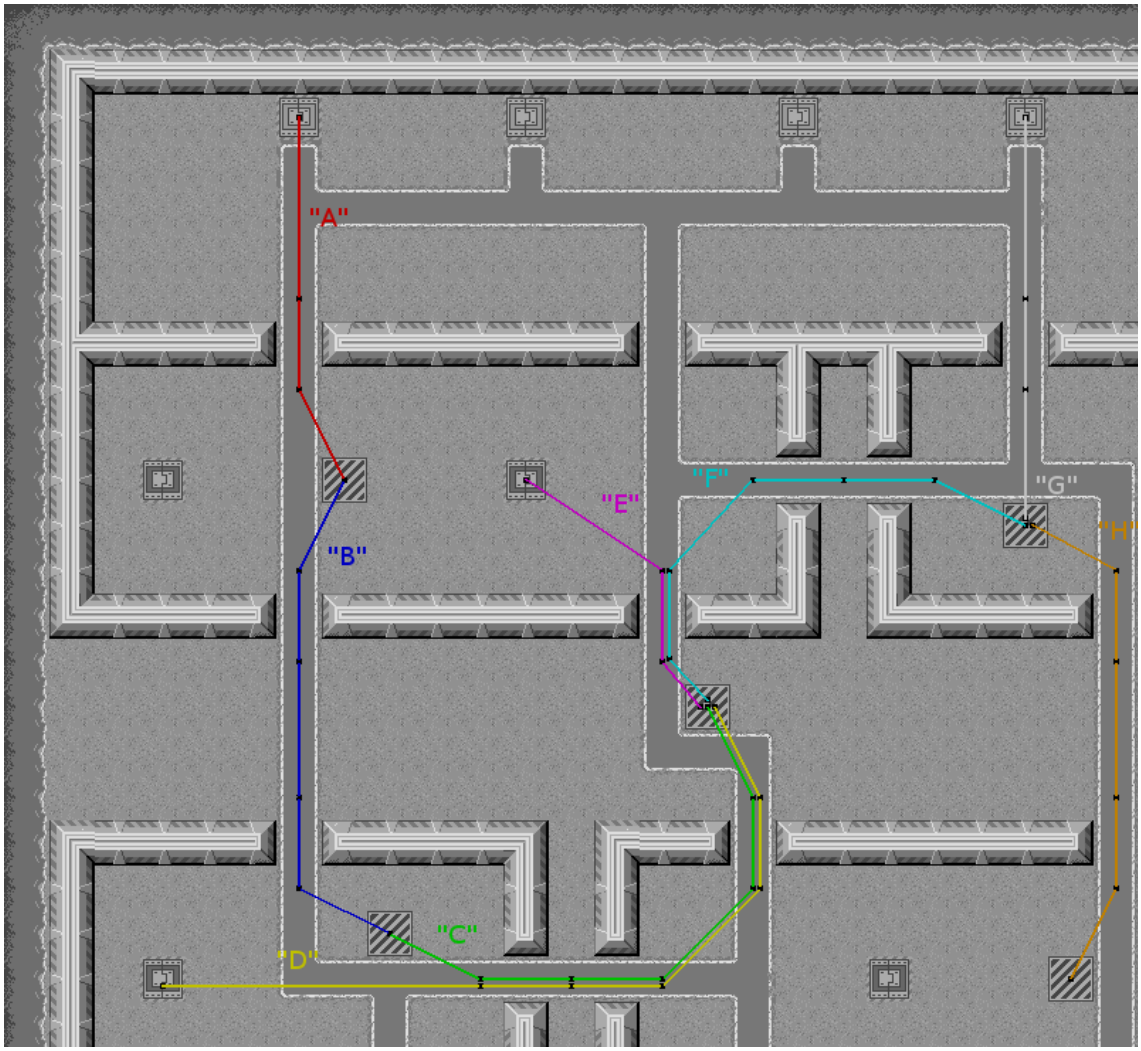
W pierwszym podejściu za heurystykę przyjęto koszt w prostej linii między danym węzłem a celem, co nie działało zbyt dobrze – heurystyka powinna być dolnym ograniczeniem kosztu, zaś takie podejście zawyżało prognozę kosztu całej drogi nie biorąc pod uwagę omijania przeszkód zachodzącego w jej dalszej części. Ostatecznie za heurystykę przyjęto prostokątny dystans między danym węzłem a celem, co przyniosło znajdowanie przewidywanych dróg.

Kod algorytmu jest identyczny jak w przypadku algorytmu Dijkstry, zmieniona jest tylko linia odpowiedzialna za priorytet węzła do odwiedzenia:

```
UWORD uwPriority = uwCost
  + ABS(pNextNode->fubX - pDstNode->fubX) + ABS(pNextNode->fubY - pDstNode->fubY);
```

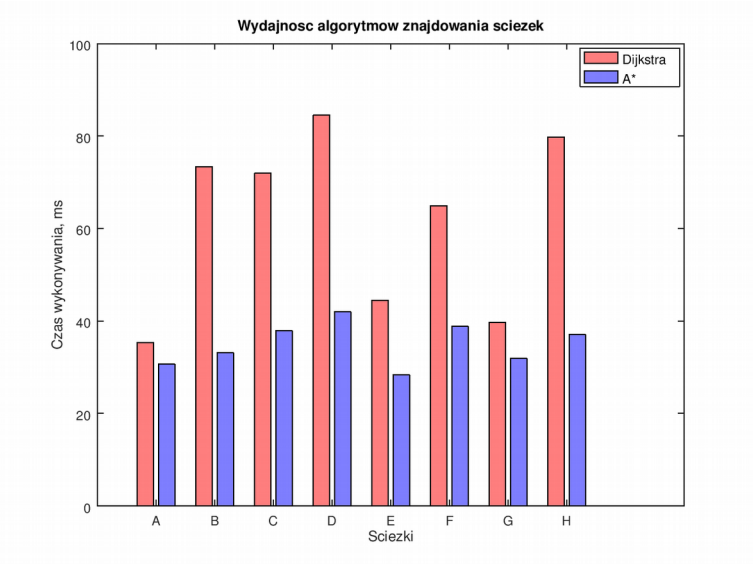
Porównanie wydajności

Oba algorytmy zastosowano na wybranych ośmiu parach „źródło”-„cel” w celu pomiaru wydajności. Pomiary są tym bardziej miarodajne, gdyż oba algorytmy zwróciły dokładnie te same trasy.



Ryc. 6. Ścieżki użyte w pomiarze wydajności algorytmów trasujących.
Źródło: opracowanie własne, GIMP 2.8

Zgodnie z przewidywaniami wydajność algorytmu A* jest w każdym przypadku większa niż algorytmu Dijkstry. Wyniki zaprezentowano na wykresie poniżej.



Ryc. 7. Zestawienie wydajności algorytmów: Dijkstry i A* na wybranych ścieżkach.
Źródło: opracowanie własne, GNU Octave 4.2.1

Dostosowanie do możliwości obliczeniowych komputera Amiga

Uzyskane czasy algorytmu A* są nawet 40-krotnie zbyt duże niż dopuszczone przez założenia projektowe. Z tego względu funkcję przepisano tak, by jej wykonywanie trwało co najwyżej przez zadany czas. Podejście takie wymagało przechowania danych niezbędnych do obliczeń w dedykowanej strukturze:

```
typedef struct _tRoute {
 UBYTE ubNodeCount; ///< Liczba węzłów na ścieżce.
 UBYTE ubCurrNode; ///< Aktualny węzeł, do którego zmierza pojazd.
 tAiNode *pNodes[ASTAR_ROUTE_NODE_MAX]; ///< Pierwszy indeks to cel, ostatni to początek
} tRoute;

typedef struct {
 UBYTE ubState; ///< Stan obliczeń
 tHeap *pFrontier; ///< Kopiec binarny trzymający kolejne pola z frontu do przeliczenia
 tAiNode *pCameFrom[AI_MAX_NODES]; ///< Z którego węzła dotrzeć na dany węzeł
 UWORD pCostSoFar[AI_MAX_NODES]; ///< Koszt od początku drogi do danego węzła
 tAiNode *pNodeDst; ///< Docelowy węzeł drogi
 tAiNode *pNodeCurr; ///< Obecnie przeliczany węzeł drogi
 UWORD uwCurrNeighbourIdx; ///< Indeks obecnie rozpatrywanego sąsiada przeliczanego węzła
 tRoute sRoute; ///< Docelowa droga
} tAstarData;
```

Dane te inicjowane są funkcją astarStart():

```
void astarStart(tAstarData *pNav, tAiNode *pNodeSrc, tAiNode *pNodeDst) {
 heapClear(pNav->pFrontier); // Wyczyść kopiec binarny frontu
 memset(pNav->pCostSoFar, 0xFF, sizeof(UWORD)*AI_MAX_NODES); // Zerowanie kosztów
 memset(pNav->pCameFrom, 0, sizeof(tAiNode*) * AI_MAX_NODES); // Zerowanie poprzedników
 pNav->pCostSoFar[pNodeSrc->fubIdx] = 0;
 pNav->pNodeDst = pNodeDst; // Ustaw węzeł docelowy
 heapPush(pNav->pFrontier, pNodeSrc, 0); // Rozpocznij przeszukiwanie od węzła startowego
 pNav->ubState = ASTAR_STATE_LOOPING; // Przejdź do stanu LOOPING w funkcji astarProcess()
 pNav->uwCurrNeighbourIdx = g_fubNodeCount; // Ostatni indeks sąsiada – spowoduje pobranie
                                            // nowego węzła z pFrontier w astarProcess()
}
```

Następnie, w momencie przeliczania zachowania danego pojazdu wywoływana jest funkcja astarProcess(), której wartość zwracana mówi o tym, czy docelowa droga została znaleziona: zero jeśli obliczenia dalej trwają, 1 gdy pole sRoute zostało wypełnione informacją o nowej drodze. Treść funkcji astarProcess zamieszczono poniżej:

```
UBYTE astarProcess(tAstarData *pNav) {
 // Komputer Amiga pozwala na pomiar czasu z dokładnością zależną od zegara obrazu PAL lub
 // NTSC, dla PAL: 1 = 0.4us => 10000 = 4ms => 2500 = 1ms
 const ULONG ulMaxTime = 2500; // liczba „tyknięć” odpowiadająca 1ms
 if(pNav->ubState == ASTAR_STATE_LOOPING) { // Kolejne przebiegi pętli obliczeń
  ULONG ulStart = timerGetPrec(); // Pobierz aktualną liczbę „tyknięć”
  do {
   if(pNav->uwCurrNeighbourIdx >= g_fubNodeCount) { // Sprawdzono wszystkich sąsiadów
    pNav->pNodeCurr = heapPop(pNav->pFrontier); // Pobierz kolejne pole do przeliczenia
    if(pNav->pNodeCurr == pNav->pNodeDst) { // Jeśli kolejne pole jest celem
     pNav->ubState = ASTAR_STATE_DONE;      // To droga została znaleziona
     return 0;
    }
    pNav->uwCurrNeighbourIdx = 0; // A jeśli nie to zresetuj licznik sprawdzonych sąsiadów
   }
   tAiNode *pNextNode = &g_pNodes[pNav->uwCurrNeighbourIdx]; // Pobierz kolejnego sąsiada.
   if(pNextNode != pNav->pNodeCurr) { // Upewnij się że sąsiad nie jest tym samym polem.
    UWORD uwCost = pNav->pCostSoFar[pNav->pNodeCurr->fubIdx] // Koszt drogi od początku
     + aiGetCostBetweenNodes(pNav->pNodeCurr, pNextNode);  // do sąsiada.
    if(uwCost < pNav->pCostSoFar[pNextNode->fubIdx) {
     // Jeśli jest niższy niż z innych pól to jest to optymalna droga do sąsiada.
     pNav->pCameFrom[pNextNode->fubIdx] = pNav->pNodeCurr; // Zastąp mniej optymalną drogę.
     pNav->pCostSoFar[pNextNode->fubIdx] = uwCost;
     UWORD uwPriority = uwCost                        // Im mniejszy koszt tym większy
      + ABS(pNextNode->fubX - pNav->pNodeDst->fubX)  // priorytet danego kierunku
      + ABS(pNextNode->fubY - pNav->pNodeDst->fubY); // poszukiwań drogi do celu.
     heapPush(pNav->pFrontier, pNextNode, uwPriority); // Dodaj sąsiada do kolejki.
    }
   }
   ++pNav->uwCurrNeighbourIdx;
  } while(timerGetDelta(ulStart, timerGetPrec()) <= ulMaxTime); // Powtarzaj dopóki jest czas
  return 0; // Obliczenia nadal się nie zakończyły
```

```
 }
 else {
  // ASTAR_STATE_DONE - zebranie trasy do struktury sRoute
  pNav->sRoute.pNodes[0] = pNav->pNodeDst; // Pierwszy jest węzeł docelowy
  pNav->sRoute.ubNodeCount = 1;
  // Pobieranie kolejnych węzłów trasy poprzez cofanie się z węzła końcowego
  tAiNode *pPrev = pNav->pCameFrom[pNav->pNodeDst->fubIdx]; // Pierwszy poprzednik
  while(pPrev) {
   pNav->sRoute.pNodes[pNav->sRoute.ubNodeCount] = pPrev; // Dodaj poprzednika do trasy
   ++pNav->sRoute.ubNodeCount;
   pPrev = pNav->pCameFrom[pPrev->fubIdx]; // Pobierz następny węzeł
  }
  pNav->sRoute.ubCurrNode = pNav->sRoute.ubNodeCount-1; // Rozpocznij od najmłodszego węzła
  pNav->ubState = ASTAR_STATE_OFF; // Algorytm zakończył pracę
  return 1; // Trasa gotowa do wykorzystania
 }
}
```

Dalszy rozwój

Poza problemami omówionymi w sprawozdaniu projektowym, istnieje wiele punktów zaczepienia, które pozwoliłyby na ulepszenie zachowania sztucznej inteligencji.

Odświeżanie kosztów i tras przy zniszczeniu terenu

Przejęcie bazy, zniszczenie elementu muru lub jednej z wieżyczek powoduje, że koszty przejazdu przez okoliczne kafle planszy ulegają zmianie. Aktualizacja tego rodzaju danych jest lokalna i łatwa do naniesienia.

Selektywne odświeżanie kosztów transportu na odcinkach między węzłami jest już bardziej problematyczne – w rozsądnym czasie możliwe jest wyłącznie zgrubne oszacowanie, czy kafel mógł być częścią ścieżki między węzłami – na podstawie sprawdzenia, czy jego współrzędne znajdują się między współrzędnymi węzła początkowego oraz końcowego.

Korekcja tras wykonywanych w danej chwili przez pojazdy na planszy wymagałoby względnie dużej ilości mocy obliczeniowej, która musiałaby zostać przeznaczona na sprawdzenie, które trasy faktycznie zależą od zmienionych warunków na planszy. Jako że trasy wyznaczone w oparciu o stare dane są wciąż poprawne, stwierdzono że trasowanie w oparciu o nowe informacje powinno dotyczyć wyłącznie przyszłych, nie obecnych tras.

Siatka nawigacyjna

Obecnie, zarówno w tworzeniu gier jak i w robotyce, odchodzi się od stosowania punktów nawigacyjnych w formie bezpośredniej – zamiast tego plansze dzieli się na strefy przy pomocy wielokątów wypukłych, z czego przeważnie stosuje się czterokąty. Podejście to jest trochę bardziej wymagające pod względem zasobów komputera, lecz pozwala na bardziej naturalne zachowanie pojazdów takie jak „ścianie” zakrętów, poruszanie się wzdłuż ścian oraz omijanie innych pojazdów w obrębie dozwolonych stref przemieszczania się.